FAULT DIAGNOSIS TOOLBOX
– v0.13

ERIK FRISK <frisk@isy.liu.se>
Department of Electrical Engineering
Linköping University, Sweden

SUMMARY
Fault Diagnosis Toolbox is a Matlab toolbox for analysis and design of fault diagnosis systems for dynamic systems, primarily described by differential equations. In particular, the toolbox is focused on techniques that utilize structural analysis, i.e., methods that analyze and utilize the model structure. The model structure is the interconnections of model variables and is often described as a bi-partite graph or an incidence matrix. Key features of the toolbox are

- Defining diagnosis models, using only model structure or full symbolic expressions.
- Diagnosability analysis - analyze a given model to determine which faults that can be detected and which faults that can be isolated
- Model exploration and analysis, e.g., plotting model properties, Dulmage-Mendelsohn decomposition, DAE index analysis, ...
- Finding overdetermined sets of equations (MSO sets), which are minimal submodels that can be used to design fault detectors
- Sensor placement - determine minimal sets of sensors needed to be able to detect and isolate faults
- Code generation (Matlab and C) for residual generators. Two different types of residual generators are supported, sequential residual generators based on a matching in the model structure graph, and observer based residual generators.

The toolbox relies on the object-oriented functionality of the Matlab language and is freely available under a MIT license. The latest version can always be downloaded from our website at http://www.fs.isy.liu.se/Software/FaultDiagnosisToolbox/ and links to relevant publications can be found also at our list of publications http://www.fs.isy.liu.se/Publications.
CONTRIBUTORS

The following people has contributed with code

- Erik Frisk, Department of Electrical Engineering, Linköping University, Sweden.

- Mattias Krysander, Department of Electrical Engineering, Linköping University, Sweden.

- Daniel Jung, Department of Electrical Engineering, Linköping University, Sweden.
## CONTENTS

1 Introduction and overview
   1.1 Reference literature ........................................ 4
   1.2 Downloading and installation ............................... 5
   1.3 Terms of usage ............................................. 6

2 Defining models
   2.1 Defining a structural model ............................... 7
      2.1.1 Defining the model using incidence matrices .......... 7
      2.1.2 Defining the model using variable names ............ 7
      2.1.3 Defining dynamic models .............................. 9
   2.2 Defining symbolic models ................................ 11
      2.2.1 Conditional constraints ............................... 12
      2.2.2 Using external functions ............................. 12

3 Dulmage-Mendelsohn decomposition ........................... 12

4 Analysis of overdetermined equations ....................... 13

5 Diagnosability analysis ....................................... 15

6 Test Selection
   6.1 A minimal hitting set approach ............................ 16
   6.2 Random Forest based Approach ............................ 16

7 Sensor placement analysis .................................... 16

8 Residual generator design .................................... 18
   8.1 Sequential residual generator design .................... 18
   8.2 Generating and compiling generated C code ............. 21
   8.3 Observer based residual generator design ............... 21

9 Use case
   9.1 Model definition ........................................... 24
   9.2 Isolability analysis ........................................ 25
   9.3 Find overdetermined set of equations .................... 26
   9.4 Design residual generators ................................ 27
   9.5 Isolability properties of residual generators .......... 28
   9.6 Simulation results ......................................... 29

A Summary of class methods .................................... 32

B Compile C++ implementations ................................. 34
   B.1 Minimal Hitting Set ........................................ 34
   B.2 MSO algorithm .............................................. 34

C Generated code in use-case .................................. 36
   C.1 ResGen1 .................................................... 36
   C.2 ResGen2 .................................................... 37
   C.3 ResGen3 .................................................... 38
   C.4 ResGen4 .................................................... 39

D Index of keywords and methods .............................. 41
1 INTRODUCTION AND OVERVIEW

This toolbox covers a set of methods and functionality for fault diagnosis of dynamic systems described by differential (or static) equations. The field of fault diagnosis is wide and there are many available methods described in the literature. This toolbox focuses on techniques from the Automatic Control community (SafeProcess) and some from the AI field (DX). In particular, techniques related to structural analysis is covered because they are particularly suited to automate in a computer tool. This manual is not intended as a book on diagnosis or structural methods and some of the material covered requires knowledge outside of this text. See Section 1.1 for some pointers to relevant literature.

This manual will not cover all the details, options, and outputs of all available methods, instead it will cover typical uses. The outline of the manual is that in Section 2, it will be covered how to define models, and then in Sections 3-8, different analysis and design techniques included in the toolbox will be covered. Section 9 describes a use case, beginning with a model definition and all the way to simulation of automatically generated residual generators. The directory examples in the source distribution includes a number of use cases, including the one covered in Section 9.

The toolbox requires Matlab v7.6 (R2008a) or later for the object-oriented functionality and for symbolic math functionality, the Symbolic Toolbox version 7.14 (2012a) or later is required\(^1\). The fault diagnosis toolbox utilizes the object oriented functionality of the Matlab language and the main class is DiagnosisModel. The most detailed documentation of the methods can be found in Matlab and to start the help browser, write

\[
\text{>> doc DiagnosisModel}
\]

It is also possible to list all available methods by

\[
\text{>> methods DiagnosisModel}
\]

Methods for class DiagnosisModel:

- AddEquations
- AddSensors
- AlgebraicVariables
- BipartiteToLaTeX
- CompiledMHS
- CompiledMSO
- DetectabilityAnalysis
- DiagnosisModel
- DifferentialConstraints
- DynamicVariables
- FSM
- GenSimulationModel
- IsDynamic
- IsHighIndex
- IsLowIndex
- IsObservable
- IsPSO
- IsStatic
- IsolabilityAnalysis
- IsolabilityAnalysisArts
- IsolabilityAnalysisFSM
- Matching
- MeasurementEquations
- ObserverResGen
- Pantelides
- Pantelides
- PlotDM
- PlotMatching
- PlotModel
- PossibleSensorLocations
- Redundancy
- RemoveFaultVariables
- ReplaceEquations
- SensorLocationsWithFaults
- SensorPlacementDetectability
- SensorPlacementIsolability
- SeqResGen
- StateVariables
- Structural
- SubModel
- TestSelection
- copy
- srank

\(^1\) The toolbox is primarily developed and tested under v8.4 (R2014b) and later versions of Matlab/Symbolic Toolbox.
Methods of DiagnosisModel inherited from handle.

To obtain help for a particular method, here for example the PlotDM method, write

```matlab
>> help DiagnosisModel.PlotDM
PlotDM Plots Dulmage–Mendelsohn decomposition of model structure

[row,col,psodecomp] = model.PlotDM( options )

Plots a Dulmage–Mendelsohn decomposition, originally described in
Dulmage, A. and Mendelsohn, N. "Coverings of bipartite graphs." 

By default, the Dulmage-Mendelsohn decomposition is plotted for the
incidence matrix for the unknown variables.

Options can be given as a number of key/value pairs

Key Value
eqclass If true, perform canonical decomposition of M+ and
plot equivalence classes

For further details on the canonical decomposition
of the M+ part of the structure, see Chapter 4 in
"Design and Analysis of Diagnosis Systems Using Structural 

fault If true, indicates fault equations in canonical 
decomposition of M+

submodel Array of equation indices corresponding to submodel.

Outputs:
row − row permutation used in the plot
col − column permutation used in the plot
psodecomp − result of psodecomposition of the M+ part

Example:
model.PlotDM( 'eqclass', true, 'fault', true )
```

1.1 Reference literature

Our publications on structural methods (all should not be included): [17, 18, 6, 10, 5, 11, 15, 16]
Other (include more): [1]

1.2 Downloading and installation

The latest version of the package can always obtained from http://www.fs.isy.liu.se/Software/FaultDiagnosisToolbox/ and the installation is very simple:
1. Uncompress the tar.gz/zip-file

2. Add the src directory to the Matlab-path

3. (optional) There are C++ implementations of some computationally expensive algorithms. It is not necessary to compile these for the toolbox to work, there are Matlab implementations directly installed. However, with the compiled versions there might be significant increases in performance.

For linux and Mac OS X there are pre-compiled versions and to install, copy the suitable files from the binaries directory into the src directory. See Appendix B for details how to compile C++ sources.

In the archive there is also a manual (this document) and a directory with a few example usages of toolbox functionality. The toolbox requires Matlab v7.6 (R2008a) or newer and for full functionality it requires access to the symbolic math toolbox.

### 1.3 Terms of usage

The toolbox is free for anyone to use, it is distributed under the MIT License (MIT) ([http://opensource.org/licenses/MIT](http://opensource.org/licenses/MIT)). If you encounter bugs, have comments, or suggestions, please contact Erik Frisk <frisk@isy.liu.se>. If you use the toolbox in your scientific work, please cite the toolbox (not yet published) and the corresponding methodological publication. The relevant publication is indicated in the help text for the class methods and functions.

### 2 Defining Models

A first step in using the toolbox is to define the model object. There are two types of model specifications

- **structural model**
- **symbolic model**

A structural model only contains information about model structure and does not need specifications on the underlying symbolic expressions. Many of the analysis methods can be applied to structural models and it is mainly the residual generation methods that need the symbolic expressions. When defining a symbolic model, the toolbox automatically computes the model structure.

To illustrate, the following small example will be used.

\[
\begin{align*}
e_1 & : x_1 = -c_1 x_1 + x_2 + x_5 \\
e_2 & : x_2 = -c_2 x_2 + x_3 + x_4 \\
e_3 & : x_3 = -c_3 x_3 + x_5 + f_1 + f_2 \\
e_4 & : x_4 = -c_4 x_4 + x_5 + f_3 \\
e_5 & : x_5 = -c_5 x_5 + u + f_4 \\
e_6 & : y_1 = x_1 \\
e_7 & : y_2 = x_2 \\
e_8 & : y_3 = x_3
\end{align*}
\]

The variables \( x_i \) are the unknown states, \( y_i \) measurement signals, \( u \) known control input, \( f_i \) the faults, and \( c_i \) known model parameters. The model structure is then given by Table 1. Here, the dynamics are lumped, i.e., the variables are considered as signals and it is not important if a variable is differentiated or not. See [5] for some further discussion on this and for dynamic models, see Section 2.1.3 for further discussion where it is shown how to explicitly define non-lumped dynamic models.
2.1 Defining a structural model

There are two ways of defining a structural model; either the incidence matrices are given directly or the variable names for each equation are specified.

2.1.1 Defining the model using incidence matrices

To define this model structure in the toolbox, i.e., creating the model object, the first important function call is DiagnosisModel. To use this function, define a structure with the model specification and then call DiagnosisModel with the structure as argument. The model specification has 4 important fields

- **type** - when specifying a model using the incidence matrices, this should be the string MatrixStruc
- **X** - incidence matrix for the unknown variables
- **F** - incidence matrix for the faults
- **Z** - incidence matrix for the known variables

In Matlab, this becomes

```matlab
modelDef.type = 'MatrixStruc';
modelDef.X = [1 1 0 0 1; 1 1 0 0 1; 0 1 0 1 0; 0 0 1 0 0; 0 0 0 1 1; ... 0 0 0 0 0; 1 0 1 0 0; 0 0 0 1 0;];
modelDef.F = [0 0 0 0; 0 0 0 0; 0 1 0 1; 0 0 1 0;];
modelDef.Z = [0 0 0 0; 0 0 0 0; 0 0 0 0; 0 0 0 1;];
```

The variable names are by default $x_i$, $z_i$ and $f_i$. To specify the variable names explicitly, add field names $x$, $z$, or $f$ respectively. For example, to specify the known variable names as in the model (1), add

```matlab
modelDef.z = {'y1','y2','y3','u'};
```

After the model specification is done, create the model object by running

```matlab
model = DiagnosisModel(modelDef);
model.name = 'Example model';
```

where also a name for the model can be specified (optional).

2.1.2 Defining the model using variable names

Defining incidence matrices is prone to errors and a more convenient way to define model the model structure is by only providing variable names. Again, a model structure is defined with 5 important fields
2.1 Defining a structural model

- **type** - when specifying a model using the incidence matrices, this should be the string 'VarStruc'
- **x** - cell array with unknown variable names
- **f** - cell array with fault variable names
- **z** - cell array with known variable names
- **rels** - a cell array describing the variables in each equation.

For the model (1), this model specification, which is equivalent to the model definition above, becomes:

```matlab
modelDef.type = 'VarStruc';
modelDef.x = {'x1','x2','x3','x4','x5'};
modelDef.z = {'y1','y2','y3','u'};
modelDef.f = {'f1','f2','f3','f4'};
modelDef.rels = {{'x1','x2','x5'},
                 {'x2','x3','x4'},
                 {'x3','x5','f1','f2'},
                 {'x4','x5','f3','f4','u'},
                 {'y1','x1'},
                 {'y2','x2'},
                 {'y3','x3'}};
model = DiagnosisModel(modelDef);
model.name = 'Example model';
```

Now that the model is defined, we can try a few simple operations on the model object. For example, the model structure can be plotted using the class method `PlotModel`. The command

```matlab
model.PlotModel()
```

will result in Figure 1. The class method `Lint` does some basic validity check on the model definition, e.g.,

```matlab
>> model.Lint()
Model: Example model

Variables
5 unknown variables
4 known variables
```
4 fault variables
8 equations, including 0 differential constraints
Degree of redundancy: 3

Model validation finished with 0 errors and 0 warnings

The model structure can also be illustrated using a bi-partite graph, the class method BipartiteToLaTeX generates \LaTeX code generating a figure that can be directly typeset using the \LaTeX-engine. For example

```matlab
model.BipartiteToLaTeX('bipartite.tex', 'faults', true, 'shortnames', true);
```

generates code for Figure 2.

As a final note it is important to understand that the DiagnosisModel class is a handle class in Matlab. This means that the variable `model` above is a reference to the object. Thus, making

```matlab
model2=model; % Warning! No new object
```
does not make a copy of the object, it merely stores another reference to the same object. To get a new copy, use the `copy` class method as

```matlab
model2 = model.copy(); % Safe, new object created
```

### Defining dynamic models

To make systematic analysis of some dynamic properties of the model, there is a need to explicitly state the dynamic variables. The way to do this is in the toolbox is to introduce some new variables and explicit differential-constraints. Thus, for model (1), introduce variables $dx_i$ for the differentiated variables, and add 5

---

differential constraints, one for each state-variable, explicitly connecting the differentiated variable with the non-differentiated. Thus, a model description, equivalent to (1), is

\[ e_1 : \frac{dx_1}{dt} = -c_1 x_1 + x_2 + x_5 \]
\[ e_2 : \frac{dx_2}{dt} = -c_2 x_2 + x_3 + x_4 \]
\[ e_3 : \frac{dx_3}{dt} = -c_3 x_3 + x_5 + f_1 + f_2 \]
\[ e_4 : \frac{dx_4}{dt} = -c_4 x_4 + x_5 + f_3 \]
\[ e_5 : \frac{dx_5}{dt} = -c_5 x_5 + u + f_4 \]
\[ e_6 : y_1 = x_1 \]
\[ e_7 : y_2 = x_2 \]
\[ e_8 : y_3 = x_3 \]

and here the differential-constraints \( e_9 \text{--} e_{13} \) is explicit. Model (2) can be defined in the toolbox using variable names just as before, but using the function DiffConstraint to define the differential constraints. Matlab code to define the dynamic model then becomes

```matlab
modelDef.type = 'VarStruc';
modelDef.x = {'dx1', 'dx2', 'dx3', 'dx4', 'dx5', 'x1', 'x2', 'x3', 'x4', 'x5'};
modelDef.z = {'y1', 'y2', 'y3', 'u'};
modelDef.f = {'f1', 'f2', 'f3', 'f4'};
modelDef.rels = {{'dx1', 'x1', 'x2', 'x5'}, {'dx2', 'x2', 'x3', 'x4'},
                 {'dx3', 'x3', 'x5', 'f1', 'f2'}, {'dx4', 'x4', 'x5', 'f3'},
                 {'dx5', 'x5', 'f4', 'u'}, {'y1', 'x1'}, {'y2', 'x2'},
                 {'y3', 'x3'},
                 DiffConstraint('dx1', 'x1'), DiffConstraint('dx2', 'x2'),
                 DiffConstraint('dx3', 'x3'), DiffConstraint('dx4', 'x4'),
                 DiffConstraint('dx5', 'x5')};
model = DiagnosisModel(modelDef);
```

Calling the `PlotModel` method results in Figure 3. This way of defining the dy-
namics is not necessarily optimal since it may introduce unnecessary variables and also may affect the index of the model. However, this forces the model to be in semi-explicit form

\[
\begin{align*}
\dot{x}_1 &= g_1(x_1, x_2) \\ 0 &= g_2(x_1, x_2)
\end{align*}
\]

which makes further analysis significantly easier. For example, the semi-explicit form (3) is (locally) low differential index [2] if and only if

\[
\frac{\partial g_2}{\partial x_2}
\]

is full rank. For a fully implicit DAE, additional analysis is needed, for example utilizing the algorithm in [13].

### 2.2 Defining symbolic models

Defining a symbolic model, i.e., specifying the symbolic expressions for the model constraints, makes it possible to make the same structural analyses as for the structural models and, in addition, generate code for residual generators. The Symbolic Math Toolbox for Matlab\(^3\) is required for this to work.

To specify the model, create a model structure with type `Symbolic` and define the model variables as before. Also add names of model parameters. In the case of model (2), this looks like

```matlab
modelDef.type = 'Symbolic';
modelDef.x = {'dx1', 'dx2', 'dx3', 'dx4', 'dx5', 'x1', 'x2', 'x3', 'x4', 'x5'};
modelDef.z = {'y1', 'y2', 'y3', 'y4', 'u'};
modelDef.f = {'f1', 'f2', 'f3', 'f4'};
modelDef.parameters = {'c1', 'c2', 'c3', 'c4', 'c5'};
```

To specify values for parameters, not necessary, include a struct with the parameter values as

```matlab
param_values.c1 = 1.0;
param_values.c2 = 2.0;
param_values.c3 = 3.0;
param_values.c4 = 4.0;
param_values.c5 = 5.0;
```

and then include in the model definition as

```matlab
modelDef.parameter_values = param_values;
```

The next step is to make all model variables and parameters symbolic. This is achieved with

```matlab
syms(modelDef.x{:})
syms(modelDef.f{:})
syms(modelDef.z{:})
syms(modelDef.parameters{:})
```

Now that all variables and parameters are symbolic, the relations of the model can be written down and the model object created as before.

```matlab
modelDef.rels = {...
    dx1 == -c1*x1+x2+x5,
    dx2 == -c2*x2+x3+x4,
}
```

The differential constraints are added, as before, using the directive `DiffConstraint`.

To tidy up, the symbolic variables can be cleared from the workspace using the commands

```matlab
% clear temporary variables from workspace
clear( modelDef.x{:} )
clear( modelDef.f{:} )
clear( modelDef.z{:} )
clear( modelDef.parameters{:} )
```

When the symbolic model is specified, the model structure is automatically computed and all the analysis/design tools utilizing the model structure can be directly applied, in addition to some new methods operating on symbolic expressions. In particular this applies to residual generation described in Section 8.

### 2.2.1 Conditional constraints

Not yet fully implemented. At the moment, define an external function as in Section 2.2.2 for this functionality.

### 2.2.2 Using external functions

It is often the case in more complex models that there are external functions like look-up tables. Such functions can be used when defining the model. For example, consider that the model has a function for \( \eta_c \) with arguments as

\[
\eta_c(\eta_{c,\text{max}}, \eta_{c,\text{min}}, X, Q)
\]

Then, defining the function to be a symbolic function as

```matlab
syms eta_c_fun(eta_cmax,eta_cmin,X,Q)
```

and then the function can be used as any other. Of course, the symbolic toolbox will not be able to do symbolic manipulations of that particular function and the causality of equations using that function will be uniquely determined by the model formulation.

### 3 Dulmage-Mendelsohn decomposition

When doing any sorts of structural analysis for fault diagnosis, the Dulmage-Mendelsohn decomposition [4] is a very useful tool [1]. Given a structural model, by proper and well defined reordering of variables and equations, a structure graph can always be transformed into the form shown in Figure 4. If \( X \) is a structure matrix, the command

```matlab
dm = GetDMParts( X );
```

is a simple wrapper around the `dmperm` command in Matlab, which computes the Dulmage-Mendelsohn decomposition. The variable `dm` is a structure with 7 fields:
For fault diagnosis, there is a particular decomposition of the overdetermined part that is of particular interest. The decomposition is defined in [11] and can be computed using the \texttt{PSODecomposition} command. There is also a class method that can plot the Dulmage-Mendelsohn decomposition of the model structure in an informative way. For this method, there are two options that can be activated, perform the decomposition of the over-determined part of the model, and indicate which equations that are influenced by faults. This is particularly important in diagnosability analysis. Below is a method call with both options activated,

\begin{verbatim}
model.PlotDM('eqclass', true, 'fault', true)
\end{verbatim}

and the result, for the three-tank model in [6], is shown in Figure 5.

4 \textbf{ANALYSIS OF OVERDETERMINED EQUATIONS}

Overdetermined parts of a model is highly interesting for fault diagnosis since these are the parts with redundancy and thereby possible to use for fault diagnosis. In the toolbox, there are two main class methods: \texttt{MSO} and \texttt{MTES}. The set of Minimally Structurally Overdetermined (MSO) sets of equations are subset minimal sets of equations with redundancy. Implemented in the toolbox is the algorithm from [11]. It is straightforward to apply. Given a model object \texttt{model}, the command
msos = model.MSO();

the set of all MSO sets are computed. Please beware that the cardinality of the set of MSO sets is exponential in the degree of redundancy of the model. Therefore, when the redundancy gets high enough the computational complexity of the algorithm becomes very high. The notion of MSO is related to other works concerning overdetermined sets of equations, see [12] for further discussions and similarities. Instructions to install compiled C++ version of the MSO algorithm for increased performance can be found in Appendix B.2.

For the model (2), there are 11 MSO sets where one for example is MSO₁ = \{3, 5, 8, 11, 13\} which means that the equations

\[
\begin{align*}
    e₃ : & \quad dx₃ = -c₃x₃ + x₅ + f₁ + f₂  \\
    e₅ : & \quad dx₅ = -c₅x₅ + u + f₄  \\
    e₈ : & \quad y₃ = x₃
\end{align*}
\]

are overdetermined and can be used to design a residual generator. For example, based on the equations above one can derive the ARR

\[
r = \dot{y₃} + (c₃ + c₅)\dot{x₃} + c₃c₅y₃ - u
\]

or the observer based residual generator

\[
\begin{align*}
    \dot{x₃} & = -c₃\dot{x₃} + \dot{x₅} + K₁(y₃ - \dot{x₃})  \\
    \dot{x₅} & = -c₅\dot{x₅} + u + K₂(y₃ - \dot{x₃})  \\
    r & = y₃ - \dot{x₃}
\end{align*}
\]

where \(K₁\) and \(K₂\) are observer gains to ensure observer stability.

As mentioned above, the size of the set of MSO sets is exponential in the redundancy of the model and therefore may not be applicable to high-redundancy problems. For this reason, a second type of overdetermined sets of equations might be of interest, Minimal Test Equation Support (MTES), defined in [9]. These do not have the as severe complexity issues as the MSO sets and the class method is called in a similar way using the method MTES.
5 DIAGNOSABILITY ANALYSIS

A set of methods for analyzing diagnosability of a model or a set of residual generators are available. Here, diagnosability means to analyze which faults that are structurally detectable and structurally isolable. Basic definitions on detectability and isolability used in the toolbox can be found in [10, 6].

For a basic detectability analysis of a given model, use the class method DetectabilityAnalysis as

```
[df,ndf] = model.DetectabilityAnalysis();
```

The df output is the set of detectable faults and ndf the set of non-detectable faults.

Similarly, to plot a fault isolability analysis of the model use the class method IsolabilityAnalysis as

```
model.IsolabilityAnalysis( );
```

With no output arguments, the method plots the analysis. It is possible to restrict the analysis to causality assumptions [6] which here means that the analysis can be done in derivative causality, integral causality, or mixed causality. The mixed causality is the default if no causality assumption is specified. To explicitly specify the causality assumption, write

```
model.IsolabilityAnalysis( ‘causality’, ‘der’);
model.IsolabilityAnalysis( ‘causality’, ‘int’);
model.IsolabilityAnalysis( ‘causality’, ‘mixed’);
```

See Figures 10 and 11 for example outputs. The interpretation is that with a non-zero element at position \((i,j)\) means that fault in column \(j\) can not be isolated from fault in row \(i\).

It is also possible to do analysis on a set of ARRs, represented as sets of equations to be used to design residual generators. For example, to see what is the isolability properties of a diagnosis system based on MSO 1 and 3 (just example numbers), use the class method IsolabilityAnalysisARRs as

```
msos = model.MSO();
model.IsolabilityAnalysisARR( msos([1,3]) );
```

It is also possible to obtain the fault sensitivity matrix (FSM) using the class method FSM as

```
FSM = model.FSM( Msos([1,3]) );
```

and then perform the analysis on the fault signature matrix using the class method IsolabilityAnalysisFSM as

```
model.IsolabilityAnalysisFSM( FSM );
```

6 TEST SELECTION

With a set of MSOs, or the corresponding fault signature matrix (FSM), it is an interesting problem how to select a subset of tests that achieves required fault isolability
performance. In general, not all possible tests are needed and often substantially less. The toolbox currently supports a simple minimal hitting set based approach to selecting tests and a data-driven approach based on Random Forest machine learning classifiers.

### 6.1 A minimal hitting set approach

The minimal hitting-set approach is implemented in the class method `TestSelection`, see [17] for further discussion on this approach. The following call finds all subset minimal sets of tests, based on the fault signature matrix $FSM$, such that maximal fault isolability is possible.

```plaintext
1 ts = model.TestSelection(FSM);
```

It is also possible to use the set of MSOs directly

```plaintext
1 ts = model.TestSelection(msos);
```

The above problem has poor complexity properties and can very quickly become intractable and therefore other methods are available. For example, to choose an approximate hitting set approach called `aminc`, which will finish fast but not guarantee a minimal solution, call

```plaintext
1 ts = model.TestSelection(msos, 'method', 'aminc');
```

### 6.2 Random Forest based Approach

The minimal hitting-set approach is implemented, not in a class method since the test selection is not tightly coupled to a model but instead a data set. Instead, the method is implemented in the `RandomForestTestSelection`, see [7] for details. In the example suite there is the engine data example from [7] implemented. For further details, and a Python implementation, see also the public git repository https://gitlab.liu.se/erifr93/MLTestSelection.

### 7 SENSOR PLACEMENT ANALYSIS

Sensor placement, or possible sensor selection, is the task of choosing a set of sensors such that diagnosis specifications are possible to reach. This toolbox implements the methods in [10].

A first step, before any analysis is possible, the set of possible sensor locations must be defined. As a principle of the approach, sensors measure single variables among the unknown variables. Thus, as is common, some sensors measure a function of the unknown variables $x$, and possibly known variables $z$, add a new variable and equation to the model

$$x_{new} = f(x)$$

Then, $x_{new}$ is the new possible sensor location. Possible sensor locations is specified using the class method `PossibleSensorLocations`. For example, below it is specified that the first 15 unknown variables in the model is possible sensor locations.

```plaintext
1 model.PossibleSensorLocations(model.x(1:15));
```

Sensor locations can be specified by name, as above, or just simply indices into the set of unknown variables. For example, if positions 1, 2, 5, and 7 are possible sensor locations, use

```plaintext
1 model.PossibleSensorLocations([1, 2, 5, 7]);
```
Also these new sensors may fail and if we want to include that into the analysis, there is a need to specify sensor locations where added sensors may fail. This is done using the class method `SensorLocationsWithFaults`. For example, if all new sensors may fail, use

```matlab
model.SensorLocationsWithFaults( model.x );
```

Now that possible sensor locations have been specified, to compute all minimal sensor sets that achieves detectability of the faults, use the method `SensorPlacementDetectability` as

```matlab
sDet = model.SensorPlacementDetectability();
sDet{:}
ans =
'Pz' 'Q'
ans =
'Pz' 'Qv' 'Qv3'
```

In this case, there are two minimal solutions where the first one is to measure the variables \( \{P_z, Q\} \) and the second is \( \{P_z, Q_v, Q_v3\} \).

If we want to add the first set of sensors, call `AddSensors`

```matlab
model.AddSensors( sDet{1} );
```

This command modifies the model `model`. In case you want a new object, with the new sensors, without modifying the original model, write

```matlab
model2 = model.AddSensors( sDet{1} );
```

The isolability properties, as described in Section 5, after adding the detectability sensors is shown in Figure 6. It is clear that all faults are detectable, but the isolation performance is far from ideal.

If the isolability in Figure 6 is not sufficient, we can instead call the class method `SensorPlacementIsolability` to find sets of sensors that not only detects fault but also makes fault isolation possible. For the same example as above, the Matlab call is

```matlab
sIsol = model.SensorPlacementIsolability();
model3 = model.AddSensors( sIsol{1} );
```

The example is taken from paper [10] and `Damadics.m` can be found in the examples directory.
There are 6 solutions, each involving 5 sensors.

```matlab
>> sIsol

sIsol =
{1x5 cell} {1x5 cell} {1x5 cell} {1x5 cell} {1x5 cell}
```

Here, again, the first solution is added to the model and again, using the diagnos-
ability analysis methods from Section 5, results in Figure 7. Here it is clear that,
except for faults entering the model in the same equations (\{f_1, f_5\}, \{f_4, f_{11}\}, and
\{f_9, f_{16}\}), full isolability is achieved.

![Isolability matrix for 'Damadics benchmark'](image)

(a) Isolability matrix

![Dulmage-Mendelsohn decomposition](image)

(b) Dulmage-Mendelsohn decomposition

Figure 7: Isolability properties after adding isolability sensors.

8 RESIDUAL GENERATOR DESIGN

In the literature, there are many different proposed approaches for residual gen-
eration, e.g., based on parity relations, Extended Kalman Filters, adaptive/high-
gain/sliding-observers and so on. In this toolbox, two basic approaches are imple-
mented to generate residual generators that are general enough to be automatic and
supported by structural analysis. It should be emphasized that these are not to be
accepted as the only or best solution, just two approaches that are particularly well
suited for automatic code generation. The first is here called sequential residual
generation and the second is a differential-algebraic observer technique suitable for
low-index problems.

8.1 Sequential residual generator design

A sequential residual generator, although the name may not be standard, the basic
approach is well known. The basic idea is that, given an overdetermined set of
equations, find a computational sequence for the unknown variables, and then ver-
ify consistency of the set of equations and observations by inserting the variables
into the residual equations. Since dynamic systems are studied, questions arise
on how to deal with differential constraints, and in this framework there are two
ways; either you integrate or you differentiate. The code generated here can be
in so called derivative causality, integral causality, or mixed causality. If there are
algebraic loops in the computational sequence, the toolbox will use the equation solving capabilities of the Symbolic Math Toolbox in Matlab. Works that describes the basic procedure are [6, 16].

To describe the basic steps of the approach, consider an overdetermined set of equations

\[ g_i(x, z, f) = 0, \quad i = 1, \ldots, n \]

A first step is to partition the set of equations into an exactly determined, with respect to the unknown variables \( x \), part and a residual equation part as

\[ g^1_i(x, z, f) = 0, \quad i = 1, \ldots, n_1 \]
\[ g^r_i(x, z, f) = 0, \quad i = 1, \ldots, n_r \]

The exactly determined part \( g^1 \) is then used to solve for \( x \) and compute \( \hat{x} \), which is then inserted into the residual equation to compute a residual as

\[ r = g^r(\hat{x}, z, f) = 0 \]

If the model is dynamic, the computational sequence might include differentiations, integrations or both. This is referred to as the sequential residual generator is in derivative, integral, or mixed causality.

The toolbox supports this design methodology using the class methods `Matching` and `SeqResGen`. The method `Matching` computes a computational sequence given an exactly determined set of equations. Then, `SeqResGen` is called given a matching and residual equations to generate the code. The examples below generates Matlab code buyt the toolbox also supports generating C code that is Matlab callable. See Section 8.2 for som further details on C code generation and compiling. As an example, consider the case where the model is an MSO set, i.e., a minimally structurally overdetermined set of equations. This means that by subtracting any equation, what is left is an exactly determined set of equations. To generate a residual, where the first equation in an MSO set is used as residual equation, and the rest is used to compute the unknown variables, the following code can be used

\begin{verbatim}
Gamma = model.Matching(setdiff(mso,mso{1})); % compute matching model.SeqResGen( Gamma, mso{1},'ResGen' );
\end{verbatim}

which will create the file `ResGen.m`, implementing the residual generator. The generated code corresponds to the function

\[ \begin{bmatrix} r_t, x_{t+1} \end{bmatrix} = \text{ResGen}(z_t, x_t, \theta, 1/fs) \]

where \( z_t \) and \( x_t \) are the known signals and residual generator state at time point \( t \), \( \theta \) the model parameters, and \( f_s \) the sampling rate. See Appendix C.1 for an example from the use case in Section 9. To use the generated function to compute a residual, based on measurements \( z \), something like this is used

\begin{verbatim}
for k=1:N
    [r(k), state] = ResGen( z(k,:), state, params, 1/fs );
end
\end{verbatim}

The state consists of information about variables \( xi \) that are numerically integrated and variables \( xd \) that are numerically differentiated according to

\[ \text{state}_t = (xi_t, xd_{t-1}) \]

Basic structure of the generated code is shown in the pseudo-code below

\begin{verbatim}
function r_t = ResGen( z_t, state_t, params, T_s )
% Initialize parameters
\theta = params.\theta;
\end{verbatim}
% Initialize integral state variables
(xi_t, xd_{t-1}) = state

% Residual generator body, sequential computations
x_1_t = ...
...
x_d_t = \frac{d}{d\tau} (\cdot)
...
x_n_t = ...

% Compute residual value
r_t = ...

% Integrate state variables
x_{i+1} = x_i + \int_{t}^{t+T_i} x'(\tau) \, d\tau

% Update state variables
state = (x_{i+1}, x_d_t)
end

See the use case in Section 9 for further details.

The causality of the residual generator is an important property. For an MSO set with n equations, with 1 more equation than unknown variables, and where each subset of n - 1 equations is exactly determined, there are n possible sequential residual generators. Although they are based on the same set of equations, they might have dramatically different properties. One such, important, property is the causality of the residual generator. To investigate, there is a convenience function that automatically computes the causality of the sequential residual generator for each choice of residual equation. The class method MSOCausalitySweep is called as

\texttt{model.MSOCausalitySweep( mso )}

which will output \texttt{der}, \texttt{int}, \texttt{mixed}, or \texttt{algebraic} for each case. For example, consider an MSO set consisting of 6 equations. A sample output of \texttt{MSOCausalitySweep} is then

\texttt{>> model.MSOCausalitySweep( mso )
\texttt{'int' 'mixed' 'der' 'mixed' 'der' 'int'}}

This means that using the first equation as a residual equation and the remaining 5 would lead to a sequential residual generator in integral causality. Using the second as residual equation would result in a mixed causality residual generator and so on.

It is possible to explicitly specify how the residual equation shall be interpreted in case it is a differential constraint. The options are derivative and integral and corresponds to the alternatives

\[
r = x - \int x' \, dt, \quad r = x' - \frac{d}{dt} x
\]

where \(x\) and \(x'\) are the variable and the corresponding computed derivative. In Matlab, the key \texttt{‘diffres’} is given to the method \texttt{MSOCausalitySweep} as

\texttt{model.MSOCausalitySweep( mso, ‘diffres’,’int’ )
\texttt{'int' 'mixed' 'der' 'mixed' 'der' 'int'}}

There is also the possibility to ask for a boolean variable indicating if it is possible to realize a residual generator for a given MSO set in derivative or integral causality respectively. The call looks like
8.2 Generating and compiling generated C code

The toolbox supports generating C code that can be compiled using the mex functionality in Matlab. The call to generate C code is similar to generating Matlab code, only passing the language option to the `SeqResGen` as

```matlab
model.SeqResGen( Gamma, mso{1}, 'ResGen', 'language', 'C');
```

This will generate the file `ResGen.cc` that can be compiled with any C++ compiler\(^5\). To compile in Matlab, write

```matlab
>> mex ResGen.cc
Building with 'Xcode Clang++'.
MEX completed successfully.
```

This function can be used exactly the same way as the Matlab counterpart. The function generated as above, and in Matlab mode, computes the residual for one sample at a time. In C mode, a batch function can also be generated. In the batch mode, not only observations at one time-point is given to the residual generator, but an entire dataset. The computational speedup is generally significant. To generate the batch version of the residual generator, write

```matlab
model.SeqResGen( Gamma, mso{1}, 'ResGenb', 'language', 'C', 'batch', true );
```

and the file `ResGenb.cc` is compiled exactly the same way as above.

If the model has external functions, as described in Section 2.2.2, these functions has to be implemented also in C code, compiled, and linked with the generated residual generator code. For example, say tat all external functions are implemented in a file `externalfuns.cc` with a corresponding header file `externalfuns.h`. Then, with the `external` option, the generated code will include the external function headers.

```matlab
model.SeqResGen( Gamma, mso{1}, 'ResGen', 'language', 'C', ...
            'external', 'externalfuns.h' );
```

Then, compiling and linking the whole thing is then done for example by

```matlab
>> mex ResGen.cc externalfuns.cc
Building with 'Xcode Clang++'.
MEX completed successfully.
```

8.3 Observer based residual generator design

This approach aims at generating code for an observer that estimates the unknown variables and computes a residual. The residual generator will be formulated as a DAE, which can be integrated using any standard ODE solver which means that it is only applicable to low-index problems \([14, 8]\). To describe the method, partition the unknown variables \(x\) into \(x_1\) which are state variables, and \(x_2\) which are algebraic variables. Then, the model can be described by

\[
\begin{align*}
    g_i(d x_1, x_1, x_2, z, f) &= 0 & i &= 1, \ldots, n \\
    d x_1 &= \frac{d}{dt} x_1 & i &= 1, \ldots, m
\end{align*}
\]  

(4)

\(^5\) The current implementation utilizes some C++ functionality and therefore the file need to be compiled with a C++ compiler.
where the dynamic relations has been explicitly described. Important note, the implemented approach is only applicable to models of low (structural) differential index [8], i.e., state-space models and implicit state-space models. This restriction is important and note that if MSO sets, or other submodels, are considered when generating residuals, the low-index property is not necessarily fulfilled even though the original model is in state-space form. Loosely, the low-index property corresponds to that there exists, locally, unique solutions for the highest ordered derivatives in $g_i$. Let $g = (g_1, \ldots, g_n)$, then the model (4) is of low index at $x = x_0$ and $z = z_0$ if

$$
\left( \frac{\partial g}{\partial x_1}, \frac{\partial g}{\partial x_2} \right) \bigg|_{x=x_0, z=z_0}
$$

has full column rank. In the toolbox, structural low index is verified which corresponds to that there exists a complete matching of the highest ordered derivatives in the equations $g$.

$$
\dot{x}_1 = g_1(x_1, x_2, z, f) \\
0 = g_2(x_1, x_2, z, f), \quad \frac{\partial g_2}{\partial x_2} \text{ is full column rank} \\
0 = g_r(x_1, x_2, z, f)
$$

In the toolbox, a test on if the model is structurally high-index or not using the call `IsHighIndex`

```matlab
model.IsHighIndex()
```

which tests if there exists a low-index submodel or not. To test if a specified submodel, e.g., an MSO set is high-index, write

```matlab
model.IsHighIndex( mso )
```

where `mso` is a vector of equation indices. The method `IsHighIndex` determines if the model has low structural differential index, i.e., 0 or 1. To determine the exact structural index, as could be interesting to analyze in high-index problems, then the method `Pantelides` from [13] can be directly applied.

```matlab
structIndex = model.Pantelides()
```

Note that method `Pantelides` can only be called on exactly determined systems.

For a low-index system, equations (5) is used to form the DAE observer, with a feedback gain $K(x, z)$ introduced as

$$
\dot{\hat{x}}_1 = g_1(\hat{x}_1, \hat{x}_2, z) + K(x, z)g_r(\hat{x}_1, \hat{x}_2, z) \\
0 = g_2(\hat{x}_1, \hat{x}_2, z)
$$

This observer estimates the unknown states $x_1$ and the algebraic variables $x_2$. Then, the residual equations in $g_r$ from (5) can be used to compute the residual.

$$
r = g_r(\hat{x}_1, \hat{x}_2, z)
$$

To generate code, suitable to be integrated using any solver that is suitable for low-index DAEs, is generated using the class method `ObserverResGen`

```matlab
model.ObserverResGen( mso, 'ResGen' );
```

This call will generate the file `ResGen.m`, see Appendix C.4 for an example from the use case in Section 9. Two possible types of integrators that might be used are solvers based on Backward Differentiation Formula (BDF) or implicit Runge-Kutta (IRK) methods [2]. Possible Matlab solvers are `ode15s` and `ode23t`. 
Let an extended state vector be \( w = (\dot{x}_1, \dot{x}_2, r) \), and the dimensions for \( \dot{x}_1, \dot{x}_2, \) and \( r \) respectively be \( n_1, n_2, \) and \( n_r \). The generated code corresponds to the function \( F(w, z) \) in the DAE model

\[
M \dot{w} = \begin{pmatrix}
g_1(\dot{x}_1, \dot{x}_2, z) + K(\dot{x}_1, \dot{x}_2, z) g_r(\dot{x}_1, \dot{x}_2, z) \\
g_2(\dot{x}_1, \dot{x}_2, z) \\
r - g_r(x_1, x_2, z)
\end{pmatrix} = F(w, z)
\] (6)

where the mass matrix \( M \) is given by

\[
M = \begin{pmatrix}
I_{n_1} & 0_{n_1 \times (n_2 + n_r)} \\
0_{(n_2 + n_r) \times n_1} & 0_{(n_2 + n_r) \times (n_2 + n_r)}
\end{pmatrix}
\]

A DAE model in the form

\[
M \dot{w} = f(w)
\]

where the mass matrix \( M \) can be integrated using stiff, implicit ODE solvers. For example, the standard Matlab ODE solver \( \text{ode}15s \) can be directly used.

As an example of a function call to use the generated residual generator, let \( z \) and \( t \) be the observations and corresponding time stamps and let \( K \) be a constant observer gain. Then the residual generator can be simulated by:

```matlab
M = [eye(2) zeros(2,4); zeros(4,6)];
[-w] = ode15s(@(ts,x) ResGen( x, interp1(t,z,ts), K, params ), ... t, xo, odeset('Mass',M));
```

where, in this case, \( n_1 = 2, n_2 = 3, \) and \( n_r = 1. \)

As with any feedback system, the feedback gain need to be determined to ensure estimator stability. In general, this is a difficult problem but the toolbox can provide some guidance. By supplying operating point and known variables, linearization matrices can be automatically computed. Let

\[
A_{ij} = \left. \frac{\partial g_i}{\partial x_j} \right|_{x=x_0, \; z=z_0}, \quad i, j = 1, 2
\]

\[
C_j = \left. \frac{\partial g_r}{\partial x_j} \right|_{x=x_0, \; z=z_0}, \quad j = 1, 2
\]

Let the estimation error be \( e = x_1 - \dot{x}_1 \), then the linearized error dynamics of the observer is \( \dot{e} = (A - KC)e \) with

\[
A = (A_{11} - A_{12}A_{22}^{-1}A_{21})
\]

\[
C = -(C_1 - C_2A_{22}^{-1}A_{21})
\]

The low-index property of the model ensures that matrix \( A_{22} \) is invertible. To obtain the matrices \( A \) and \( C \), add options `linpoint` and `parameters` to the method call like the following

```matlab
[A,C] = model.ObserverResGen( mso, 'ResGen', 'linpoint', linpoint, ... 'parameters', params );
```

See further details in the use case in Section 9.4 for examples on how to form `linpoint` and `parameters`, and also how to compute a locally stabilizing feedback gain \( K \).

9 USE CASE

This section shows a simple use case how the toolbox can be used. As an example model, the three-tank model from [6] is used. All code is available, and possible to
run, in file usecase.m found in the examples/ThreeTankSimulation/ directory. The same usecase, but with C-generated residual generators is found in file usecase_c.m in the same directory. The three-tank system is shown in Figure 8 and simple model

![Diagram of the three-tank system.](image)

Figure 8: Diagram of the three-tank system.

of the system is

\[ e_1 : q_1 = \frac{1}{R_{V1}} (p_1 - p_2) \quad e_7 : y_1 = p_1 \]
\[ e_2 : q_2 = \frac{1}{R_{V2}} (p_2 - p_3) \quad e_8 : y_2 = q_2 \]
\[ e_3 : q_3 = \frac{1}{R_{V3}} (p_3) \quad e_9 : y_3 = q_0 \]
\[ e_4 : \dot{p}_1 = \frac{1}{C_{T1}} (q_0 - q_1) \quad e_{10} : \dot{p}_1 = \frac{d p_1}{d t} \]
\[ e_5 : \dot{p}_2 = \frac{1}{C_{T2}} (q_1 - q_2) \quad e_{11} : \dot{p}_2 = \frac{d p_2}{d t} \]
\[ e_6 : \dot{p}_3 = \frac{1}{C_{T3}} (q_2 - q_3) \quad e_{12} : \dot{p}_3 = \frac{d p_3}{d t} \]

where \( p_i \) is the pressure in tank \( i \), \( q_i \) the flow through valve \( i \), \( R_{Vi} \) the flow resistance of valve \( i \), and \( C_{Ti} \) the capacitance of tank \( i \). Three sensors \( y_1, y_2, \) and \( y_3, \) measure \( p_1, q_2, \) and \( q_0, \) respectively. For this study, six parametric faults have been considered in the plant: change in capacity of tanks \( C_{T1}, C_{T2}, \) and \( C_{T3}, \) and partial blocks in valves \( R_{V1}, R_{V2}, R_{V3}. \)

### 9.1 Model definition

Here, the model will be defined using symbolic expressions. Therefore, the model is defined using the following Matlab code

```matlab
modelDef.type = 'Symbolic';
modelDef.x = {'p1', 'p2', 'p3', 'q0', 'q1', 'q2', 'q3', 'dp1', 'dp2', 'dp3'};
modelDef.f = {'fV1', 'fV2', 'fV3', 'fT1', 'fT2', 'fT3'};
modelDef.z = {'y1', 'y2', 'y3'};
modelDef.parameters = {'Rv1', 'Rv2', 'Rv3', 'CT1', 'CT2', 'CT3'};

syms(modelDef.x);
syms(modelDef.f);
syms(modelDef.z);
syms(modelDef.parameters);

modelDef.rels = [q1==1/Rv1*(p1-p2) + fV1,... % e1
                q2==1/Rv2*(p2-p3) + fV2, ... % e2
                q3==1/Rv3*p3 + fV3,... % e3
                dp1==1/CT1*(q0-q1) + fT1,... % e4
```
9.2 Isolability analysis

Since sensors already have been added in the model definition, one next step is to see what kind of isolability properties that are possible (structurally). To do isolability analysis, as described in [6], the class method `IsolabilityAnalysis` can be used. Here, it is possible to show what isolability that is possible using derivative, integral, or mixed causality residual generators. First, to look at the derivative and integral causality cases, the commands

```matlab
model.IsolabilityAnalysis(' causality ',' der ');
model.IsolabilityAnalysis(' causality ',' int ');
```

produces Figures 10-a and b. See [6] for details on how to interpret the figures.

Figure 11-a shows the full structural isolability properties of the model, i.e., performance in mixed causality. The figure tells us that it is possible to 1) uniquely isolate faults $f_{V1}$, $f_{T1}$, and $f_{T2}$, and 2) the group of faults $\{f_{V2}, f_{V3}, f_{T3}\}$ can be detected and isolated from the other faults, but can not be separated from each other. Figure 11-b shows the corresponding Dulmage-Mendelsohn decomposition, with indication of faults and canonical decomposition of the overdetermined part of the model. In this case, the model only consist of an overdetermined part. The following commands in Matlab produces the figures

```matlab
model = DiagnosisModel( modelDef );
model.name = 'Three tank system';
model.PlotModel();
```
9.3 Find overdetermined set of equations

Let’s say we are happy with the isolability performance in Figure 11, a next step is to design residual generators. One way to do this is to find overdetermined set of equations and use those to design residual generators. For this, we compute the set of MSO sets, i.e., the set of minimally structurally overdetermined sets of equations, see [11] for full details on how this is done. The Matlab command

```
msos = model.MSO();
```
Design residual generators

A next step is to use these overdetermined sets of equations to generate residuals. For demonstration purposes, 4 different designs will be made. The first three are sequential residual generators, one in derivative causality (r1), one in integral causality (r2), and one in mixed causality (r3). The forth residual generator (r4) will be designed using a simple observer based approach. Residual generators r1, r2 and r4 will use MSO set $M_2$ in (7) and r3 will use $M_1$.

The first residual generator is corresponds to Fig. 2 in [6], that use equation $e_5$ as a residual equation and the remaining, exactly determined, equations in $M_2$ to compute the unknown variables. Figure 12 shows the corresponding computational graph. It is clear that the residual generator is in derivative causality, since all differential constraints (indicated in red in the figure) computes the differentiated variable from the non-differentiated. To generate code for the residual generator, first the matching is found using all equations but $e_5$, then the Matlab code is generated based on this matching. The corresponding Matlab code is

```matlab
% Gamma1 = model.Matching(setdiff(msos[2],5)); % compute matching
model.SeqResGen( Gamma1, 5, 'ResGen1');
```

and the generated code is shown in Appendix C.1.

If using $e_7$ instead of $e_5$ as a residual equation, with the same MSO set, we obtain a residual generator in integral causality instead. The corresponding computational graph, same as Fig. 3 in [6], is shown in Figure 13. It is clear that the residual generator is in integral causality, since all differential constraints computes the integral of a differentiated variable. Code generation is done as before,

```matlab
% Gamma2 = model.Matching(setdiff(msos[2],7)); % compute matching
model.SeqResGen( Gamma2, 7, 'ResGen2');
```

and the generated code is included in Appendix C.2.

To show an example of a sequential residual generator in mixed causality, i.e., where both differentiation and integration is used in the computation of the residual, consider MSO set $M_1$ with equation $e_2$ as residual equation. The computational graph is shown in Figure 14, which corresponds to Fig. 12 in [6]. Code generation is done again in the same way.
9.5 Isolability properties of residual generators

The isolability analysis in Section 9.2 was done for the model, not these four particular residual generators. This can be done using the class methods FSM and IsolabilityAnalysisFSM. First, the fault signature matrix (FSM), i.e., which faults each residual is (structurally) sensitive to is determined using FSM, and then this fault signature matrix is used to compute the isolability properties using IsolabilityAnalysisFSM. The results are shown in Figure 15 and the figures are generated using the commands:

```
Gamma3 = model.Matching(setdiff(msos{1,2})); % compute matching
del(
    model.SeqResGen( Gamma3, 2, 'ResGen3' ));

and the resulting code is shown in Appendix C.3.

In the fourth residual generator, mso M_2 is again used but now using an observer approach. Then, no residual equation need to be specified, the approach chooses residual equations automatically. Note that this approach is only applicable to low-index problems. If a linearization point and values on the parameters are provided, A and C matrices are computed such that a feedback gain can be designed to (locally) stabilize the observer. In code, this corresponds to

```
linpoint.x0 = [0,0,0,0,0,0]; linpoint.z0 = [0,0,0];
params.Rv1 = 1; params.Rv2 = 1; params.Rv3 = 1;
params.CT1 = 1; params.CT2 = 1; params.CT3 = 1;
    'parameters', params);
```

and the generated code is included in Appendix C.4. To find a feedback gain K using pole placement, a simple approach is for example

```
K = place(A',C',[−0.2,−0.3]);
```

9.5 Isolability properties of residual generators

The isolability analysis in Section 9.2 was done for the model, not these four particular residual generators. This can be done using the class methods FSM and IsolabilityAnalysisFSM. First, the fault signature matrix (FSM), i.e., which faults each residual is (structurally) sensitive to is determined using FSM, and then this fault signature matrix is used to compute the isolability properties using IsolabilityAnalysisFSM. The results are shown in Figure 15 and the figures are generated using the commands:

```
Gamma3 = model.Matching(setdiff(msos{1,2})); % compute matching
del(
    model.SeqResGen( Gamma3, 2, 'ResGen3' ));

and the resulting code is shown in Appendix C.3.

In the fourth residual generator, mso M_2 is again used but now using an observer approach. Then, no residual equation need to be specified, the approach chooses residual equations automatically. Note that this approach is only applicable to low-index problems. If a linearization point and values on the parameters are provided, A and C matrices are computed such that a feedback gain can be designed to (locally) stabilize the observer. In code, this corresponds to

```
linpoint.x0 = [0,0,0,0,0,0]; linpoint.z0 = [0,0,0];
params.Rv1 = 1; params.Rv2 = 1; params.Rv3 = 1;
params.CT1 = 1; params.CT2 = 1; params.CT3 = 1;
    'parameters', params);
```

and the generated code is included in Appendix C.4. To find a feedback gain K using pole placement, a simple approach is for example

```
K = place(A',C',[−0.2,−0.3]);
```

9.5 Isolability properties of residual generators

The isolability analysis in Section 9.2 was done for the model, not these four particular residual generators. This can be done using the class methods FSM and IsolabilityAnalysisFSM. First, the fault signature matrix (FSM), i.e., which faults each residual is (structurally) sensitive to is determined using FSM, and then this fault signature matrix is used to compute the isolability properties using IsolabilityAnalysisFSM. The results are shown in Figure 15 and the figures are generated using the commands:

```
Gamma3 = model.Matching(setdiff(msos{1,2})); % compute matching
del(
    model.SeqResGen( Gamma3, 2, 'ResGen3' ));

and the resulting code is shown in Appendix C.3.

In the fourth residual generator, mso M_2 is again used but now using an observer approach. Then, no residual equation need to be specified, the approach chooses residual equations automatically. Note that this approach is only applicable to low-index problems. If a linearization point and values on the parameters are provided, A and C matrices are computed such that a feedback gain can be designed to (locally) stabilize the observer. In code, this corresponds to

```
linpoint.x0 = [0,0,0,0,0,0]; linpoint.z0 = [0,0,0];
params.Rv1 = 1; params.Rv2 = 1; params.Rv3 = 1;
params.CT1 = 1; params.CT2 = 1; params.CT3 = 1;
    'parameters', params);
```

and the generated code is included in Appendix C.4. To find a feedback gain K using pole placement, a simple approach is for example

```
K = place(A',C',[−0.2,−0.3]);
```
9.6 Simulation results

The system can now be simulated in all fault modes. For the simulations, a simple LQ-controller is designed such that the level in tank 1 follows a reference signal. A sample, noise-free, simulation result for the no-fault case is shown in Figure 16. A simulation of the faulty case $f_{Rv1}$ is shown in Figure 17 where a ramp fault is added, starting at $t = 6$ and reaching top value at $t = 10$.

To simulate the residual generators, residuals $r_1$, $r_2$, and $r_3$ are implemented in discrete time. Let $z$ and $t$ be the observations and time stamps respectively, then simulation of the residual generators is done by

```matlab
for k=1:N
    [r1(k), state1] = ResGen1( z(k,:), state1, params, 1/fs );
    [r2(k), state2] = ResGen2( z(k,:), state2, params, 1/fs );
    [r3(k), state3] = ResGen3( z(k,:), state3, params, 1/fs );
end
```

Residual generator $r_4$ was created in continuous time, and therefore one of Matlab's standard ODE integrators are used. The code corresponds to a dynamic system in the form (6) and note that a stiff, implicit, solver is needed to integrate the
9.6 Simulation results

Figure 16: Simulation of fault free operation of the three-tank system.

Figure 17: Simulation of the faulty case, $f_{Rv1}$, of the three-tank system.
observer. In our case, we chose the solve ode15s. The code to integrate the residual generator is

```matlab
M4 = [eye(2) zeros(2,4); zeros(4,6)];
[-x] = ode15s(@(ts,x) ResGen4( x, interp1(t,z,ts), K4, params ), ...
t, xo, odeset('Mass',M4, 'AbsTol', 1e-3));
r4 = x(:,6);
```

where we use interp1 to interpolate measurement values in between sampling points. For this fault mode, residuals $r_1$, $r_2$, and $r_4$ shall indicate a fault while $r_3$ shall not, see Figure 15. Figure 18 shows the residuals. It is clear that, after the initial transient all residuals react as expected. It is also clear that $r_1$ is more noisy and that is due to that this residual is in derivative causality and the approximate differentiation introduces this error. It is also clearly visible that residual $r_1$ only reacts to changes in the fault, i.e., only reacts in the interval $[6, 10]$. Remember that residuals $r_2$ and $r_4$ are based on exactly the same set of equations as $r_1$ and this clearly shows how different ways of realizing a residual generator, based on the same set of equations has different performance and properties.

![Residual values for residuals $r_1$, $r_2$, $r_3$, and $r_4$ for the fault mode $f_{Rv1}$ shown in Figure 17.](image-url)
APPENDIX

A SUMMARY OF CLASS METHODS

Below is a table that summarizes all the class methods for DiagnosisModel objects. The description is brief and the most complete documentation can be found by starting the documentation browser using doc DiagnosisModel, or accessing the help documentation by

```
help DiagnosisModel.methodName
```

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiagnosisModel</td>
<td>Constructor for model object used for diagnosis analysis and residual generator code generation</td>
</tr>
<tr>
<td>AddSensors</td>
<td>Add sensors to a model</td>
</tr>
<tr>
<td>AddEquations</td>
<td>Add equations to a model</td>
</tr>
<tr>
<td>RemoveFaultVariables</td>
<td>Remove fault variables from a model</td>
</tr>
<tr>
<td>ReplaceEquations</td>
<td>Replace equations in a model</td>
</tr>
<tr>
<td>LumpDynamics</td>
<td>Lump dynamic variables for structural model</td>
</tr>
<tr>
<td>Structural</td>
<td>Convert a symbolic model to a structural model</td>
</tr>
<tr>
<td>copy</td>
<td>Make a new copy of the model object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model properties</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Unknown variables</td>
</tr>
<tr>
<td>f</td>
<td>Fault variables</td>
</tr>
<tr>
<td>z</td>
<td>Known variables</td>
</tr>
<tr>
<td>X</td>
<td>Incidence matrix for unknown variables</td>
</tr>
<tr>
<td>F</td>
<td>Incidence matrix for fault variables</td>
</tr>
<tr>
<td>Z</td>
<td>Incidence matrix for known variables</td>
</tr>
<tr>
<td>e</td>
<td>Equation names</td>
</tr>
<tr>
<td>name</td>
<td>Model name</td>
</tr>
<tr>
<td>type</td>
<td>Model type</td>
</tr>
<tr>
<td>P</td>
<td>List of possible sensor locations</td>
</tr>
<tr>
<td>Pfault</td>
<td>Which sensor locations may be faulty</td>
</tr>
<tr>
<td>parameters</td>
<td>List of model parameter names</td>
</tr>
<tr>
<td>syme</td>
<td>Symbolic equations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model exploration</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lint</td>
<td>Print model information and check for inconsistencies</td>
</tr>
<tr>
<td>PlotDM</td>
<td>Plots Dulmage-Mendelsohn decomposition of model structure</td>
</tr>
<tr>
<td>PlotModel</td>
<td>Plots a model object</td>
</tr>
<tr>
<td>SubModel</td>
<td>Extract submodel</td>
</tr>
<tr>
<td>MeasurementEquations</td>
<td>Extract indices to measurement equations</td>
</tr>
<tr>
<td>DifferentialConstraints</td>
<td>Extract indices to differential constraints and variables in the model</td>
</tr>
<tr>
<td>GenSimulationModel</td>
<td>Generate code for simulation model, usable directly with Matlabs ODE/DAE solvers. Model must be low-index and exactly determined.</td>
</tr>
</tbody>
</table>

continued on next page
## A summary of class methods

### continued from previous page

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ne</td>
<td>Number of equations in model</td>
</tr>
<tr>
<td>nf</td>
<td>Number of fault variables in model</td>
</tr>
<tr>
<td>nx</td>
<td>Number of unknown variables in model</td>
</tr>
<tr>
<td>nz</td>
<td>Number of known variables in model</td>
</tr>
<tr>
<td>Redundancy</td>
<td>Compute the structural degree of redundancy of a model</td>
</tr>
<tr>
<td>MTESRedundancy</td>
<td>Compute the structural degree of redundancy for an MTES set</td>
</tr>
<tr>
<td>srank</td>
<td>Compute the structural rank of the incidence matrix for the unknown variables</td>
</tr>
<tr>
<td>AlgebraicVariables</td>
<td>Extract the algebraic variables of the model</td>
</tr>
<tr>
<td>DynamicVariables</td>
<td>Extract the dynamic variables of the model</td>
</tr>
<tr>
<td>IsStatic</td>
<td>Is the set of model equations static?</td>
</tr>
<tr>
<td>IsDynamic</td>
<td>Is the set of model equations dynamic?</td>
</tr>
<tr>
<td>IsHighIndex</td>
<td>Is the model of high structural differential index?</td>
</tr>
<tr>
<td>IsLowIndex</td>
<td>Is the model of low structural differential index?</td>
</tr>
<tr>
<td>IsPSO</td>
<td>Is the model proper structurally overdetermined?</td>
</tr>
<tr>
<td>IsObservable</td>
<td>Is the (sub-)model structurally observable?</td>
</tr>
<tr>
<td>Pantelides</td>
<td>Run Pantelides algorithm for determining structural index and differentiation vector</td>
</tr>
<tr>
<td>BipartiteToLaTeX</td>
<td>Generate a LaTeX document with the bipartite graph corresponding to the structural model</td>
</tr>
</tbody>
</table>

### Diagnosability analysis

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DetectabilityAnalysis</td>
<td>Performs a structural detectability analysis</td>
</tr>
<tr>
<td>IsolabilityAnalysis</td>
<td>Perform structural single fault isolability analysis of model</td>
</tr>
<tr>
<td>IsolabilityAnalysisArrows</td>
<td>Perform structural single fault isolability analysis of a set of ARRs</td>
</tr>
<tr>
<td>IsolabilityAnalysisFSM</td>
<td>Perform structural single fault isolability analysis of a Fault Signature Matrix (FSM)</td>
</tr>
</tbody>
</table>

### Sensor placement

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PossibleSensorLocations</td>
<td>Set possible sensor locations</td>
</tr>
<tr>
<td>SensorLocationsWithFaults</td>
<td>Set possible sensor locations that has faults in new sensors</td>
</tr>
<tr>
<td>SensorPlacementDetectability</td>
<td>Determine minimal set of sensors to achieve detectability</td>
</tr>
<tr>
<td>SensorPlacementIsolability</td>
<td>Determine minimal set of sensors to achieve maximal fault isolability</td>
</tr>
<tr>
<td>CompiledMHS</td>
<td>Use the compiled minimal-hitting set algorithm if available (not recommended)</td>
</tr>
</tbody>
</table>

### Overdetermined equations

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSO</td>
<td>Compute the set of MSO sets</td>
</tr>
<tr>
<td>MTES</td>
<td>Computes the set of minimal test equation support</td>
</tr>
<tr>
<td>FSM</td>
<td>Compute the fault signature matrix (FSM)</td>
</tr>
<tr>
<td>TestSelection</td>
<td>A minimal hitting set based test selection approach</td>
</tr>
</tbody>
</table>

*continued on next page*
### Method name Description

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompiledMSO</td>
<td>Use the compiled MSO algorithm if available</td>
</tr>
<tr>
<td>ObserverResGen</td>
<td>Generate Matlab code for observer based residual generator</td>
</tr>
<tr>
<td>Matching</td>
<td>Compute a matching in the model for a set of equations</td>
</tr>
<tr>
<td>PlotMatching</td>
<td>Plot a matching</td>
</tr>
<tr>
<td>SeqResGen</td>
<td>Generate Matlab code for sequential residual generator</td>
</tr>
<tr>
<td>MSOCausalitySweep</td>
<td>For a given MSO set, determine causality for sequential residual generator</td>
</tr>
</tbody>
</table>

### Compile C++ Implementations

To compile the C++ sources, it is required that a functioning compiler is installed and configured for use with Matlab. Instructions below are directly applicable for Unix/MacOS systems. There are binaries included in the distribution if you do not want to compile the sources yourself. In the directory binaries there are compiled files for 64 bit Linux and MacOS. Copy the files corresponding to your system into the src directory and hope for the best.

#### B.1 Minimal Hitting Set

There is a compiled C++ implementation of the minimal hitting set algorithm used. To compile, open Matlab and go to src directory. To compile, type (output from a MacOS system)

```matlab
>> mex MHScompiled.cc
Building with 'Xcode Clang++'.
MEX completed successfully.
```

Verify that everything has worked by

```matlab
>> exist('MHSCompiled')
ans =
3
```

The use of the compiled algorithm is optional, full functionality is obtained with the Matlab implementation of the minimal hitting set algorithm.

#### B.2 MSO algorithm

The MSO algorithm uses a library for computing with sparse matrices. The sparse matrix library is part of the software SuiteSparse and can be downloaded from [http://faculty.cse.tamu.edu/davis/suitesparse.html](http://faculty.cse.tamu.edu/davis/suitesparse.html) and is described in [3]. You do not have to compile and install the entire SuiteSparse library, only the CSparse part. The CSparse source is included in the zip-archive.

To install on a Linux or a Mac with MacOS with developer tools installed, go to the CSparse directory and write\(^6\) at a terminal prompt, not in Matlab:

---

\(^6\) Remark for Linux users: On my Linux system, I had to add the compiler flag -fPIC on the CF line in file SuiteSparse/CSparse/Lib/Makefile.
When the CSParse library is compiled, to compile the MSO algorithm, open Mat-

> cd CSparse

lab and go to the src directory and type:

> make

```matlab
%% Specify installation directory for CSparse
CSPARSEDIR = './/CSparse';

% Derive include and lib directory
CSPARSEINC=['−I' fullfile(CSPARSEDIR, '/Include')];
CSPARSELIB=['−L' fullfile(CSPARSEDIR, '/Lib')];

%% Compile sources and link mex−file
mex('−c', '−largeArrayDims', CSPARSEINC,'MSOAlg.cc');
mex('−c', '−largeArrayDims', CSPARSEINC,'SparseMatrix.cc');
mex('−c', '−largeArrayDims', CSPARSEINC,'StructuralAnalysisModel.cc');
mex('−c', '−largeArrayDims', CSPARSEINC,'FindMSOcompiled.cc');

% Link
mex(CSPARSELIB, 'FindMSOcompiled.o', 'SparseMatrix.o', 'StructuralAnalysisModel.o', 'MSOAlg.o', '−lcsparse');
```

On a Mac system, you might get warnings when linking due to different deploy-

ment versions used by the Matlab compiler and the compiler used, “ld: warning: ob-
ject file (../CSparse/Lib/libcsparse.a(cs_add.o)) was built for newer OSX version (10.12) than
being linked (10.9)”. To avoid this warning, recompile CSParse with the build version
of Matlab by setting the environment variable MACOSX_DEPLOYMENT_TARGET=10.9 be-
fore running make to build CSParse.

Verify that everything has worked and the binary has been generated by

```matlab
>> exist ('FindMSOcompiled')
ans =
    3
```
Below are the generated code for the four residual generators in the use case in Section 9.

### c.1 ResGen1

```matlab
function [r, state] = ResGen1(z,state,params,Ts)
% RESGEN1 Sequential residual generator for model 'Three tank system'
% Causality: Der
% % Structurally sensitive to faults: fV1, fT1, fT2
% % Example of basic usage:
% % Let z be the observations and N the number of samples, then
% % the residual generator can be simulated by:
% % for k=1:N
% % [r(k), state] = ResGen1( z(k,:), state, params, 1/fs );
% % end
% % where state is a structure with the state of the residual generator.
% % The state has fieldnames: p1, p2
% File generated 29-Feb-2016 16:22:07

% Parameters
Rv1 = params.Rv1;
CT1 = params.CT1;
CT2 = params.CT2;

% Known variables
y1 = z(1);
y2 = z(2);
y3 = z(3);

% Initialize state variables
p1 = state.p1;
p2 = state.p2;

% Residual generator body
q2 = y2; %% e8
q0 = y3; %% e9
p1 = y1; %% e7
dp1 = ApproxDiff(p1,state.p1,Ts); %% e10
q1 = q0-CT1*dp1; %% e4
p2 = p1-Rv1*q1; %% e1
dp2 = ApproxDiff(p2,state.p2,Ts); %% e11
r=dp2-(q1-q2)/CT2; %% e5

% Update state variables
state.p1 = p1;
state.p2 = p2;
end

function dx=ApproxDiff(x,xold,Ts)
if length(xold)==1
    dx = (x-xold)/Ts;
elseif length(xold)==2
    dx = (3*x-4*xold(1)+xold(2))/2/Ts;
end
```

% Parameters
Rv1 = params.Rv1;
CT1 = params.CT1;
CT2 = params.CT2;

% Known variables
y1 = z(1);
y2 = z(2);
y3 = z(3);

% Initialize state variables
p1 = state.p1;
p2 = state.p2;

% Residual generator body
q2 = y2; %% e8
q0 = y3; %% e9
p1 = y1; %% e7
dp1 = ApproxDiff(p1,state.p1,Ts); %% e10
q1 = q0-CT1*dp1; %% e4
p2 = p1-Rv1*q1; %% e1
dp2 = ApproxDiff(p2,state.p2,Ts); %% e11
r=dp2-(q1-q2)/CT2; %% e5

% Update state variables
state.p1 = p1;
state.p2 = p2;
end

function dx=ApproxDiff(x,xold,Ts)
if length(xold)==1
    dx = (x-xold)/Ts;
elseif length(xold)==2
    dx = (3*x-4*xold(1)+xold(2))/2/Ts;
end
else
    error('Differentiation of order higher than 2 not supported');
end

function [r, state] = ResGen2(z,state,params,Ts)
% RESGEN2 Sequential residual generator for model 'Three tank system'
% Causality: Int
% Structurally sensitive to faults: fV1, fT1, fT2
% Example of basic usage:
% Let z be the observations and N the number of samples, then
% the residual generator can be simulated by:
% for k=1:N
%    [r(k), state] = ResGen2( z(k,:), state, params, 1/fs );
% end
% where state is a structure with the state of the residual generator.
% The state has fieldnames: p2, p1
% File generated 29-Feb-2016 16:22:09

% Parameters
Rv1 = params.Rv1;
CT1 = params.CT1;
CT2 = params.CT2;

% Known variables
y1 = z(1);
y2 = z(2);
y3 = z(3);

% Initialize state variables
p2 = state.p2;
p1 = state.p1;

% Residual generator body
q2 = y2;  %% e8
q0 = y3;  %% e9
q1 = (p1-p2)/Rv1;  %% e1
dp2 = (q1-q2)/CT2;  %% e5
dp1 = (q0-q1)/CT1;  %% e4
r=-p1+y1;  %% e7

% Update integrator variables
p2 = ApproxInt(dp2,state.p2,Ts);  %% e11
p1 = ApproxInt(dp1,state.p1,Ts);  %% e10

% Update state variables
state.p2 = p2;
state.p1 = p1;
end

function x1=ApproxInt(dx,x0,Ts)
    x1 = x0 + Ts*dx;
end
C.3 ResGen3

function [r, state] = ResGen3(z, state, params, Ts)
% RESGEN3 Sequential residual generator for model 'Three tank system'
% Causality: Mixed
%
% Structurally sensitive to faults: fV2, fV3, fT1, fT2, fT3
%
% Example of basic usage:
% Let z be the observations and N the number of samples, then
% the residual generator can be simulated by:
%
% for k=1:N
% [r(k), state] = ResGen3( z(k,:), state, params, 1/fs );
% end
%
% where state is a structure with the state of the residual generator.
% The state has fieldnames: p3, p2, p1
%
% File generated 29-Feb-2016 16:22:12

% Parameters
Rv2 = params.Rv2;
Rv3 = params.Rv3;
CT1 = params.CT1;
CT2 = params.CT2;
CT3 = params.CT3;

% Known variables
y1 = z(1);
y2 = z(2);
y3 = z(3);

% Initialize state variables
p3 = state.p3;
p2 = state.p2;
p1 = state.p1;

% Residual generator body
q2 = y2;  \% e8
q0 = y3;  \% e9
q3 = p3/Rv3;  \% e3
dp3 = (q2-q3)/CT3;  \% e6
p1 = y1;  \% e7
dp1 = ApproxDiff(p1, state.p1, Ts);  \% e10
q1 = q0-CT1*dp1;  \% e4
dp2 = (q1-q2)/CT2;  \% e5

r=q2-(p2-p3)/Rv2;  \% e2

% Update integrator variables
p3 = ApproxInt(dp3, state.p3, Ts); % e12
p2 = ApproxInt(dp2, state.p2, Ts); % e11

% Update state variables
state.p3 = p3;
state.p2 = p2;
state.p1 = p1;
end

function dx=ApproxDiff(x,xold,Ts)
if length(xold)==1
dx = (x-xold)/Ts;
elseif length(xold)==2
  dx = (3*x-4*xold(1)+xold(2))/2/Ts;
else
  error('Differentiation of order higher than 2 not supported');
end
end

function x1=ApproxInt(dx,x0,Ts)
  x1 = x0 + Ts*dx;
end

c.4 ResGen4

function dx = ResGen4(x,z,K,params)
% RESGEN4 Observer based residual generator for model 'Three tank system'
%
% Structurally sensitive to faults: fV1, fT1, fT2
%
% Example of basic usage:
% Let z and t be the observations and corresponding timestamps. Let K be the observer gain,
% then the residual generator can be simulated by:
% 
% [~,x] = ode15s(@(ts,x) ResGen4( x, interp1(t,z,ts), K, params ), t, x0, odeset('Mass',M));
% 
% where the mass matrix M is [eye(2) zeros(2,4);zeros(4,6)]
% The residual after integration is then r=x(:,6)
%
% File generated 29-Feb-2016 16:22:13
%
% Parameters
Rv1 = params.Rv1;
CT1 = params.CT1;
CT2 = params.CT2;
%
% Known variables
y1 = z(1);
y2 = z(2);
y3 = z(3);
%
% Model variables
p1 = x(1);
p2 = x(2);
q0 = x(3);
q1 = x(4);
q2 = x(5);
r1 = x(6);
%
% Algebraic equations
g21 = -q0+y3;
g22 = q1-(p1-p2)/Rv1;
g23 = -q2+y2;
%
% Residual equations
gr1 = p1+r1-y1;
%
% Dynamics, with feedback
dp1 = (q0-q1)/CT1 + K(1,:)*r1;
dp2 = (q1-q2)/CT2 + K(2,:)*r1;
% Return value
dx = [dp1; dp2; g21; g22; g23; gr1];
end
# D  INDEX OF KEYWORDS AND METHODS

## KEYWORDS

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>algebraic loops, 19</td>
<td></td>
</tr>
<tr>
<td>aminc, see minimal hitting set</td>
<td></td>
</tr>
<tr>
<td>analytical redundancy relations, see ARR</td>
<td></td>
</tr>
<tr>
<td>ARR, 15</td>
<td></td>
</tr>
<tr>
<td>causality, 20</td>
<td></td>
</tr>
<tr>
<td>derivative, see derivative causality</td>
<td></td>
</tr>
<tr>
<td>integral, see integral causality</td>
<td></td>
</tr>
<tr>
<td>mixed, see mixed causality</td>
<td></td>
</tr>
<tr>
<td>computational sequence, see matching</td>
<td></td>
</tr>
<tr>
<td>conditional constraints, 12</td>
<td></td>
</tr>
<tr>
<td>DAE model, 22</td>
<td></td>
</tr>
<tr>
<td>observer, 22</td>
<td></td>
</tr>
<tr>
<td>derivative causality, 15, 18</td>
<td></td>
</tr>
<tr>
<td>detectability analysis, 15</td>
<td></td>
</tr>
<tr>
<td>detectable faults, 15</td>
<td></td>
</tr>
<tr>
<td>diagnosability, 15</td>
<td></td>
</tr>
<tr>
<td>DiagnosisModel, 4</td>
<td></td>
</tr>
<tr>
<td>differential constraint residual equation, 20</td>
<td></td>
</tr>
<tr>
<td>differential constraints, 9, 12</td>
<td></td>
</tr>
<tr>
<td>differential index, 11, 22</td>
<td></td>
</tr>
<tr>
<td>high, 22</td>
<td></td>
</tr>
<tr>
<td>low, 22</td>
<td></td>
</tr>
<tr>
<td>structural, 22</td>
<td></td>
</tr>
<tr>
<td>dmperm, see Dulmage-Mendelsohn decomposition</td>
<td></td>
</tr>
<tr>
<td>doc, 4</td>
<td></td>
</tr>
<tr>
<td>downloading, 5</td>
<td></td>
</tr>
<tr>
<td>Dulmage-Mendelsohn decomposition, 12</td>
<td></td>
</tr>
<tr>
<td>dynamic model, 9</td>
<td></td>
</tr>
<tr>
<td>dynamics, 6</td>
<td></td>
</tr>
<tr>
<td>equivalence class, 13</td>
<td></td>
</tr>
<tr>
<td>external functions, 12, 21</td>
<td></td>
</tr>
<tr>
<td>fault sensitivity matrix, see FSM FSM, 15</td>
<td></td>
</tr>
<tr>
<td>handle class, 9</td>
<td></td>
</tr>
<tr>
<td>if-equations, 12</td>
<td></td>
</tr>
<tr>
<td>incidence matrix, 7</td>
<td></td>
</tr>
<tr>
<td>installation, 5</td>
<td></td>
</tr>
<tr>
<td>integral causality, 15, 18</td>
<td></td>
</tr>
<tr>
<td>isolability analysis, 15, 25</td>
<td></td>
</tr>
<tr>
<td>diagnosis system, 28</td>
<td></td>
</tr>
<tr>
<td>license, 6</td>
<td></td>
</tr>
<tr>
<td>lumped dynamics, 6</td>
<td></td>
</tr>
<tr>
<td>mass matrix, 23</td>
<td></td>
</tr>
<tr>
<td>matching, 19, 23</td>
<td></td>
</tr>
<tr>
<td>MatrixStruc, 7</td>
<td></td>
</tr>
<tr>
<td>methods, 4</td>
<td></td>
</tr>
<tr>
<td>mex, 34</td>
<td></td>
</tr>
<tr>
<td>MHS, see minimal hitting set</td>
<td></td>
</tr>
<tr>
<td>minimal hitting set, 16</td>
<td></td>
</tr>
<tr>
<td>minimal hitting set, 34 approximate, 16</td>
<td></td>
</tr>
<tr>
<td>minimal test equation support, see MTES</td>
<td></td>
</tr>
<tr>
<td>minimally structurally overdetermined, see MSO</td>
<td></td>
</tr>
<tr>
<td>mixed causality, 15, 18</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td></td>
</tr>
<tr>
<td>dynamic, 9</td>
<td></td>
</tr>
<tr>
<td>structure, see structural model</td>
<td></td>
</tr>
<tr>
<td>symbolic, see symbolic model, 11</td>
<td></td>
</tr>
<tr>
<td>model parameters, 11</td>
<td></td>
</tr>
<tr>
<td>MSO, 13</td>
<td></td>
</tr>
<tr>
<td>MTES, 13, 14</td>
<td></td>
</tr>
<tr>
<td>non-detectable faults, 15</td>
<td></td>
</tr>
<tr>
<td>observer gain, 23</td>
<td></td>
</tr>
<tr>
<td>ODE solver, 23</td>
<td></td>
</tr>
<tr>
<td>overdetermined equations, 13, 26</td>
<td></td>
</tr>
<tr>
<td>pole placement, 28</td>
<td></td>
</tr>
<tr>
<td>PSO decomposition, 13</td>
<td></td>
</tr>
<tr>
<td>rels, 8</td>
<td></td>
</tr>
<tr>
<td>residual equation, 19</td>
<td></td>
</tr>
<tr>
<td>residual generator, 18 observer, 21, 28 sequential, 18</td>
<td></td>
</tr>
<tr>
<td>sensor fault, 17</td>
<td></td>
</tr>
<tr>
<td>sensor placement, 16</td>
<td></td>
</tr>
<tr>
<td>sensor selection, 16</td>
<td></td>
</tr>
<tr>
<td>SeqResGen</td>
<td></td>
</tr>
<tr>
<td>batch, 21</td>
<td></td>
</tr>
<tr>
<td>C code generation, 21</td>
<td></td>
</tr>
<tr>
<td>structural model, 6, 7</td>
<td></td>
</tr>
<tr>
<td>symbolic, 11</td>
<td></td>
</tr>
<tr>
<td>symbolic model, 6</td>
<td></td>
</tr>
<tr>
<td>symbolic models, 11</td>
<td></td>
</tr>
<tr>
<td>type, 7, 8</td>
<td></td>
</tr>
<tr>
<td>VarStruc, 8</td>
<td></td>
</tr>
</tbody>
</table>
METHODS

AddEquations, 32
AddSensors, 17, 32
AlgebraicVariables, 33
BipartiteToLaTeX, 9, 33
CompiledMHS, 33
CompiledMSO, 34
copy, 9, 32
DetectabilityAnalysis, 15, 33
DiagnosisModel, 4, 7, 9, 32
DiffConstraint, 10, 12
DifferentialConstraints, 32
DynamicVariables, 33
e, 32
F, 32
f, 32
FSM, 15, 28, 33
GenSimulationModel, 32
GetDMParts, 12
IsDynamic, 33
IsHighIndex, 22, 33
IsLowIndex, 33
IsObservable, 33
IsolabilityAnalysis, 15, 25, 33
IsolabilityAnalysisArrs, 15, 33
IsolabilityAnalysisFSM, 15, 28, 33
IsPSO, 33
IsStatic, 33
Lint, 8, 32
LumpDynamics, 32
Matching, 19, 27, 34
MeasurementEquations, 32
MSO, 13, 33
MSOCausalitySweep, 20, 34
MTES, 13, 14, 33
MTESRedundancy, 33
name, 32
ne, 33
nf, 33
nx, 33
nz, 33
ObserverResGen, 22, 28, 34
P, 32
Pantelides, 22, 33
parameters, 32
Pfault, 32
PlotDM, 5, 32
PlotMatching, 34
PlotModel, 8, 10, 32
PossibleSensorLocations, 16, 33
PSODEcomposition, 13
RandomForestTestSelection, 16
Redundancy, 33
RemoveFaultVariables, 32
ReplaceEquations, 32
SensorLocationsWithFaults, 17, 33
SensorPlacementDetectability, 17, 33
SensorPlacementIsolability, 17, 33
SeqResGen, 19, 21, 27, 34
srank, 33
Structural, 32
SubModel, 32
syme, 32
TestSelection, 16, 33
type, 32
X, 32
x, 32
Z, 32
z, 32
REFERENCES


