FAULT DIAGNOSIS TOOLBOX – v0.1

ERIK FRISK[1] <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

- Finding overdetermined sets of equations (MSO sets), which are minimal submodels that can be used to design fault detectors
- Diagnosability analysis - analyze a given *model* to determine which faults that can be detected and which faults that can be isolated
- Sensor placement - determine minimal sets of sensors needed to be able to detect and isolate faults
- Code generation (Matlab) 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.

CONTENTS

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 3-7, different analysis and design techniques included in the toolbox will be covered. Section 8 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 8.

The toolbox requires Matlab v7.6 (R2008a)[1] or later. For symbolic math functionality, the Symbolic Toolbox is required. 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

```
>> doc DiagnosisModel
```

It is also possible to list all available methods by

```
>> methods DiagnosisModel

 Methods for class DiagnosisModel:

AddSensors                PlotDM
BipartiteToLaTeX          PlotModel
CompiledMHS               PossibleSensorLocations
DetectabilityAnalysis     Redundancy
DiagnosisModel            SensorLocationsWithFaults
FSM                       SensorPlacementDetectability
IsHighIndex               SensorPlacementIsolability
IsPSO                     SeqResGen
IsolabilityAnalysis       Structural
IsolabilityAnalysisArrs   SubModel
IsolabilityAnalysisFSM    TestSelection
Lint                      copy
LumpDynamics              ne
MSO                       nf
MSOCausalitySweep         nx
MTES                      nz
Matching                  srank
ObserverResGen

```

1 The toolbox is primarily developed and tested under v8.4 (R2014b).

```
 Methods of DiagnosisModel inherited from handle.
```

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

```
>> 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."
    Canadian Journal of Mathematics 10.4 (1958): 516-534.

  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
                Methods", PhD thesis, Mattias Krysander, 2006.
    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): [13, 14, 4, 7, 3, 8, 11, 12]

Other (include more): [1]

1.2 Downloading and installation

The latest version of the package can always be obtained from `http://www.fs.isy.liu.se/Software/FaultDiagnosisToolbox/` and the installation is very simple: uncompress the tar.gz/zip-file and add the src directory to the Matlab-path. 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.

There is a compiled C++ implementation of the minimal hitting set algorithm used. If you have a compiler installed and Matlab properly configured, go to the src directory In Matlab and type (output from an OS X system)

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

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

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). 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.

$$
\begin{aligned}
e_1 &: \dot{x}_1 = -c_1 x_1 + x_2 + x_5 \\
e_2 &: \dot{x}_2 = -c_2 x_2 + x_3 + x_4 \\
e_3 &: \dot{x}_3 = -c_3 x_3 + x_5 + f_1 + f_2 \\
e_4 &: \dot{x}_4 = -c_4 x_4 + x_5 + f_3 \\
e_5 &: \dot{x}_5 = -c_5 x_5 + u + f_4 \\
e_6 &: y_1 = x_1 \\
e_7 &: y_2 = x_2 \\
e_8 &: y_3 = x_3
\end{aligned}
\tag{1}
$$

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

Table 1: Model structure of example model (1).

	x_1	x_2	x_3	x_4	x_5	y_1	y_2	y_3	u	f_1	f_2	f_3	f_4
e_1	X	X			X								
e_2		X	X	X									
e_3			X		X					X	X		
e_4				X	X							X	
e_5					X				X				X
e_6	X					X							
e_7		X					X						
e_8			X					X					

differentiated or not. See [3] 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

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

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

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

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

```
sm = DiagnosisModel(model);
sm.name = 'Example model';
```

where also a name for the model is 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

- `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

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

```
sm.PlotModel()
```

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

```
>> sm.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
```

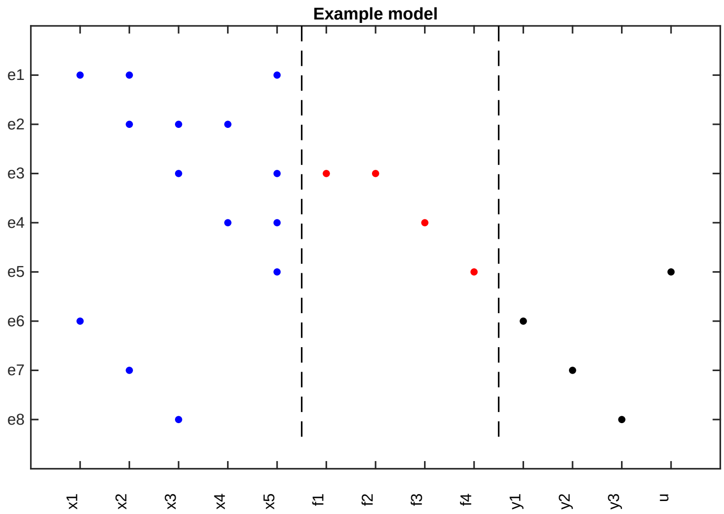


Figure 1: Result of class method `PlotModel`.

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

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

generates code for Figure 2.

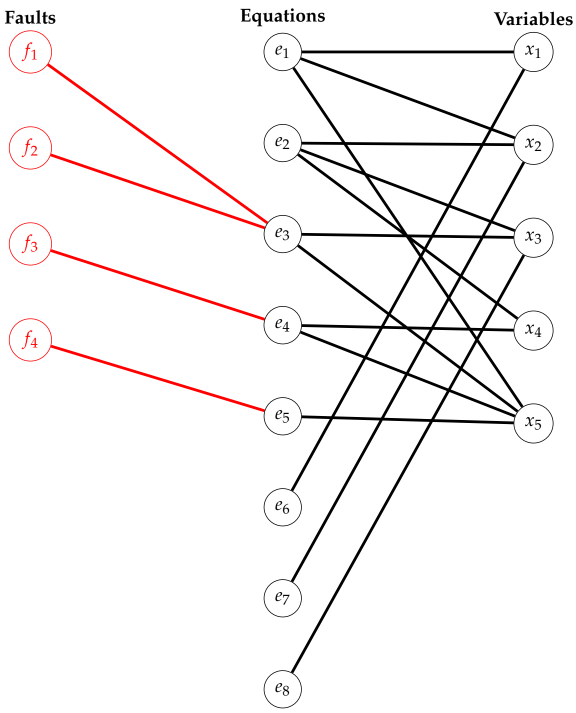


Figure 2: Result of class method `BipartiteToLaTeX`.

As a final note it is important to understand that the `DiagnosisModel` class is a *handle*[2] class in Matlab. This means that the variable `sm` above is a *reference* to the object. Thus, making

2 See http://www.mathworks.com/help/matlab/handle-classes.html for details.

```
sm2=sm; % 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

```
sm2 = sm.copy(); % Safe, new object created
```

2.1.3 *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 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

$$
\begin{aligned}
e_1 &: \ dx_1 = -c_1 x_1 + x_2 + x_5 & e_9 &: \ dx_1 = \frac{d}{dt}x_1 \\
e_2 &: \ dx_2 = -c_2 x_2 + x_3 + x_4 & e_{10} &: \ dx_2 = \frac{d}{dt}x_2 \\
e_3 &: \ dx_3 = -c_3 x_3 + x_5 + f_1 + f_2 & e_{11} &: \ dx_3 = \frac{d}{dt}x_3 \\
e_4 &: \ dx_4 = -c_4 x_4 + x_5 + f_3 & e_{12} &: \ dx_4 = \frac{d}{dt}x_4 \\
e_5 &: \ dx_5 = -c_5 x_5 + u + f_4 & e_{13} &: \ dx_5 = \frac{d}{dt}x_5 \\
e_6 &: \ y_1 = x_1 \\
e_7 &: \ y_2 = x_2 \\
e_8 &: \ y_3 = x_3
\end{aligned}
\tag{2}
$$

land here the differential-constraints e_9-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

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

Calling the `PlotModel` method results in Figure 3.

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

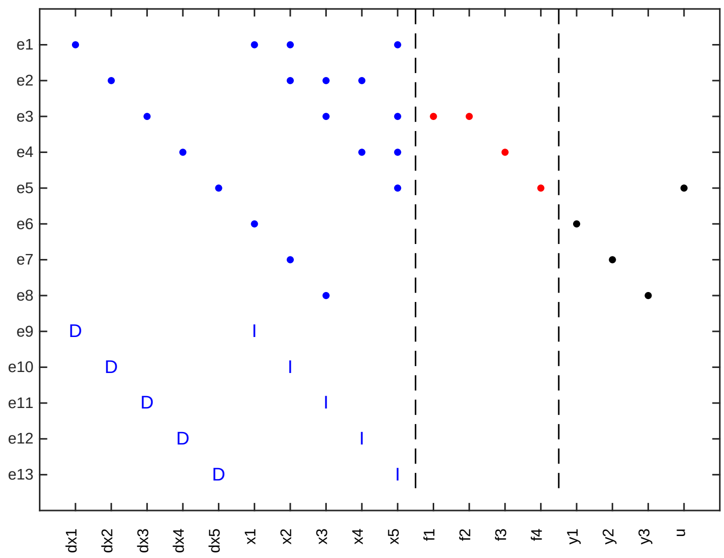


Figure 3: Result of calling the `PlotModel` method for the model (2). Note the I and D that indicate differentiated/integrated variable relation.

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

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

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

```
syms(model.x{:})
syms(model.f{:})
syms(model.z{:})
syms(model.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.

```
model.rels = {...
  dx1 == -c1*x1+x2+x5,...
  dx2 == -c2*x2+x3+x4,...
  dx3 == -c3*x3 + x5 + f1 + f2 ,...
  dx4 == -c4*x4+x5+f3,...
  dx5 == -c5*x5 + u + f4,...
  y1 == x1, y2 == x2, y3 == x3,  ...
  DiffConstraint('dx1','x1'),DiffConstraint('dx2','x2 '),...
  DiffConstraint('dx3','x3'),DiffConstraint('dx4','x4 '),...
  DiffConstraint('dx5','x5')};
```

3 http://www.mathworks.com/products/symbolic/

```
sm = DiagnosisModel( model );
sm.name = 'Example model';
```

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

```
% clear temporary variables from workspace
clear( model.x{:} )
clear( model.f{:} )
clear( model.z{:} )
clear( model.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 7.

2.2.1 *Conditional constraints*

Many models involves conditional expressions, i.e., what is referred to as if-equations in Modelica. The toolbox currently supports, for symbolic models only, conditional expressions in the form

```
if cond>=0
  expr1;
else
  expr2;
end
```

Such constraints are specified using the `IfConstraint` directive. For example, equations like below are common when modeling flow of compressible fluids past restrictions

$$\Pi_f = \begin{cases} \Pi & \Pi \geq \left(\frac{2}{\gamma+1}\right)^{\frac{\gamma}{\gamma-1}} \\ \left(\frac{2}{\gamma+1}\right)^{\frac{\gamma}{\gamma-1}} & \text{otherwise} \end{cases}$$

In the toolbox this is represented as

```
IfConstraint(PI-(2/(gamma+1))^(gamma/(gamma-1)), ... % condition
  PIf == PI, % condition true
PIf == (2/(gamma+1))^(gamma/(gamma-1))) % condition false
```

3 DULMAGE–MENDELSOHN DECOMPOSITION

When doing any sorts of structural analysis for fault diagnosis, the Dulmage-Mendelsohn decomposition [2] 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

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

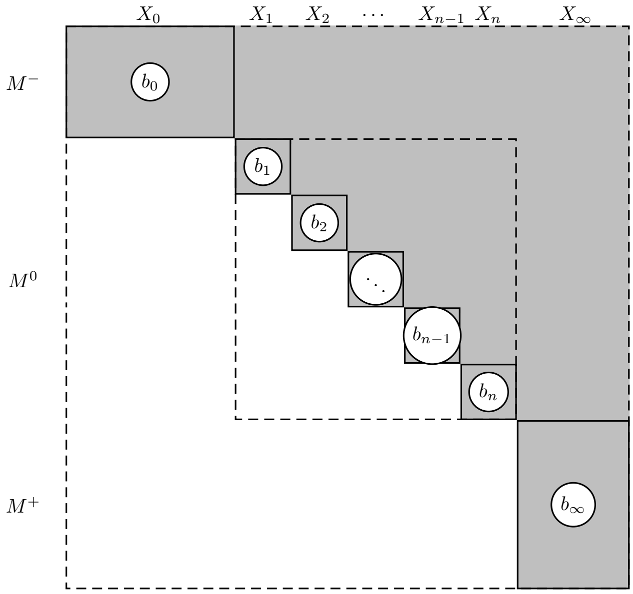


Figure 4: Dulmage-Mendelsohn decomposition

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:

- `Mm` - structure defining the rows and columns of the under-determined part M^-.
- `M0` - cell array with structures defining the Hall components in M^0
- `Mp` - structure defining the rows and columns of the over-determined part M^+.
- `M0eqs` - collection of all rows in M^0
- `M0vars` - collection of all columns in M^0
- `rowp` - original row permutation
- `colp` - original column permutation

For fault diagnosis, there is a particular decomposition of the overdetermined part that is of particular interest. The decomposition is defined in [8] and can be computed using the `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 overdetermined 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,

```
sm.PlotDM('eqclass', true, 'fault', true)
```

and the result, for the three-tank model in [4], is shown in Figure 5.

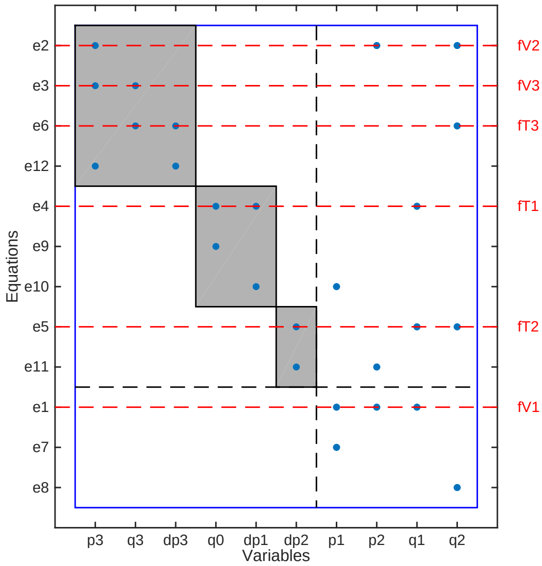


Figure 5: Dulmage-Mendelsohn decomposition, with equivalence class decomposition, and fault equation indication activated.

4 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: `MSO` and `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 [8]. It is straightforward to apply. Given a model object `sm`, the command

```
msos = sm.MSO();
```

the set of all MSOs are computed. Please beware that the cardinality of the set of MSOs 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 [9] for further discussions and similarities.

For the model (2), there are 11 MSOs where one for example is $\text{MSO}_1 = \{3, 5, 8, 11, 13\}$ which means that the equations

$$
\begin{aligned}
&e_3: \; dx_3 = -c_3 x_3 + x_5 + f_1 + f_2 \quad && e_{11}: \; dx_3 = \frac{d}{dt} x_3 \\
&e_5: \; dx_5 = -c_5 x_5 + u + f_4 && e_{13}: \; dx_5 = \frac{d}{dt} x_5 \\
&e_8: \; y_3 = x_3
\end{aligned}
$$

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

$$r = \ddot{y}_3 + (c_3 + c_5)\dot{y}_3 + c_3 c_5 y_3 - u$$

or the observer based residual generator

$$\begin{aligned}\dot{\hat{x}}_3 &= -c_3\hat{x}_3 + \hat{x}_5 + K_1(y_3 - \hat{x}_3)\\ \dot{\hat{x}}_5 &= -c_5\hat{x}_5 + u + +K_2(y_3 - \hat{x}_3)\\ r &= y_3 - \hat{x}_3\end{aligned}$$

where K_1 and K_2 are observer gains to ensure observer stability.

As mentioned above, the size of the set of MSOs are 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 [6]. 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`

```
msos = sm.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 [7, 4].

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

```
[df,ndf] = sm.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

```
sm.IsolabilityAnalysis( );
```

With no output arguments, the method plots the analysis. It is possible to restrict the analysis to causality assumptions [4] 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

```
sm.IsolabilityAnalysis( 'causality', 'der');
sm.IsolabilityAnalysis( 'causality', 'int');
sm.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 = sm.MSO();
sm.IsolabilityAnalysisARR( msos([1,3]) );
```

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

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

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

```
sm.IsolabilityAnalysisFSM( FSM );
```

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 using the class method `TestSelection`, see [13] 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.

```
ts = sm.TestSelection(FSM);
l
```

It is also possible to use the set of MSOs directly

```
ts = sm.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

```
ts = sm.TestSelection(msos, 'method', 'aminc');
```

6 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 [7].

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.

```
sm.PossibleSensorLocations(sm.x(1:15));
```

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

```
sm.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

```
sm.SensorLocationsWithFaults( sm.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

```
sDet = sm.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 $\{P_z, Q_v, Q_{v3}\}$[4]

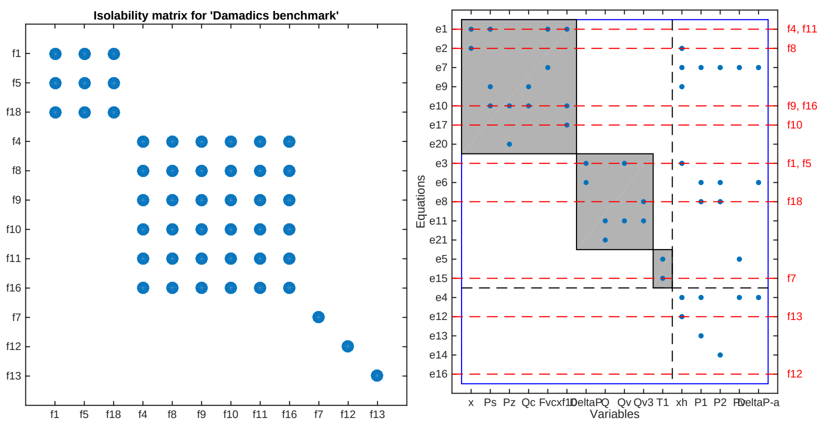
If we want to add the first set of sensors, call `AddSensors`

```
sm.AddSensors( sDet{1} );
```

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

```
sm2 = sm.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.



(a) Isolability matrix **(b)** Dulmage-Mendelsohn decomposition

Figure 6: Isolability properties after adding the detectability sensors.

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

4 The example is taken from paper [7] and `Damadics.m` can be found in the `examples` directory.

detects fault but also makes fault isolation possible. For the same example as above, the Matlab call is

```
sIsol = sm.SensorPlacementIsolability();
sm3 = sm.AddSensors( sIsol{1} );
```

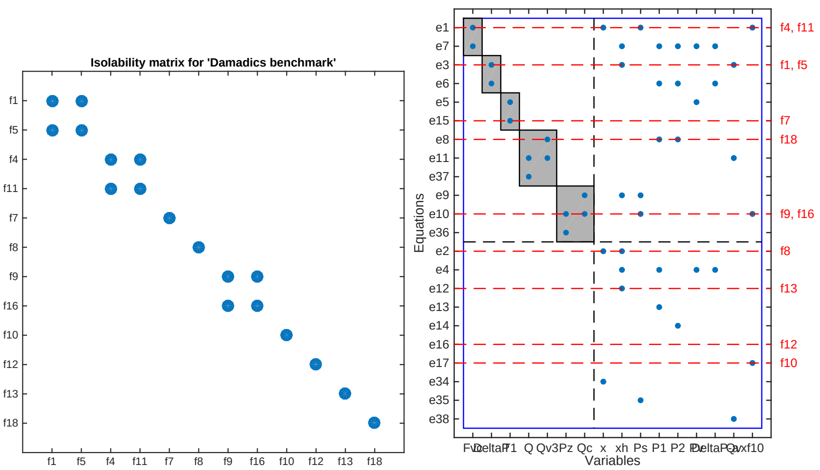
There are 6 solutions, each involving 5 sensors.

```
>> sIsol

sIsol =

  {1x5 cell} {1x5 cell} {1x5 cell} {1x5 cell} {1x5 cell} {1x5 cell}
```

Here, again, the first solution is added to the model and again, using the diagnosability 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.



(a) Isolability matrix **(b)** Dulmage-Mendelsohn decomposition

Figure 7: Isolability properties after adding isolability sensors.

7 RESIDUAL GENERATOR DESIGN

In the literature, there are many different proposed approaches for residual generation, e.g., based on parity relations, Extended Kalman Filters, adaptive/high-gain/sliding-observers and so on. In this toolbox, two basic approaches are implemented 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.

7.1 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 verify 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 [4, 12]. *reference?*

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

$$\begin{aligned} g_i^1(x, z, f) &= 0, \quad i = 1, \ldots, n_1 \\ g_i^r(x, z, f) &= 0, \quad i = 1, \ldots, n_r \end{aligned}$$

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

$$r = g_i^r(\hat{x}, z, f) = 0$$

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. As an example, consider the case where the model is an MSO, 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 is used as residual equation, and the rest is used to compute the unknown variables, the following code can be used

```
Gamma = sm.Matching(setdiff(mso,mso{1})); % coompute matching
sm.SeqResGen( Gamma, mso{1},'ResGen' );
```

which will create the file `ResGen.m`, implementing the residual generator. See Appendix B.1 for an example from the use case in Section 8. To use the generated function to compute a residual, based on measurements `z`, something like this is used

```
for k=1:N
  [r(k), state] = ResGen( z(k,:), state, params, 1/fs );
end
```

See the use case in Section 8 for further details.

The causality of the residual generator is an important property. For an MSO 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

```
sm.MSOCausalitySweep( mso )
```

which will output `Der`, `Int`, `Mixed`, or `Algebraic` for each case. For example, consider an MSO consisting of 6 equations. A sample output of `MSOCausalitySweep` is then

```
>> sm.MSOCausalitySweep( mso )
    '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 `'diffres'` is given to the method `MSOCausalitySweep` as

```
sm.MSOCausalitySweep( mso, 'diffres','Int' )
    '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 in derivative or integral causality respectively. The call looks like

```
sm.MSOCausalitySweep( mso, 'causality','Der' )
```

7.2 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 [10, 5]. 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

$$\begin{aligned} g_i(dx_1, x_1, x_2, z, f) &= 0 \qquad i = 1, \ldots, n \\ dx_1 &= \frac{d}{dt}x_1 \quad i = 1, \ldots, m \end{aligned} \tag{3}$$

where the dynamic relations has been explicitly described. Important note, the implemented approach is only applicable to models of *low (structural) differential index* [5], 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 (3) is of low index at $x = x_0$ and $z = z_0$ if

$$\left. \begin{pmatrix} \frac{\partial g}{\partial dx_1} & \frac{\partial g}{\partial x_2} \end{pmatrix} \right|_{x=x_0,\ z=z_0}$$

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.

$$\begin{aligned} \dot{x}_1 &= g_1(x_1, x_2, z, f) \\ 0 &= g_2(x_1, x_2, z, f), \quad \frac{\partial g_2}{\partial x_2} \text{ is full column rank} \\ 0 &= g_r(x_1, x_2, z, f) \end{aligned} \tag{4}$$

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

```
sm.IsHighIndex()
```

and to test if a submodel, e.g., an MSO is high-index, write

```
sm.IsHighIndex( mso )
```

where `mso` is a vector of equation indices.

For a low-index system, equations (4) is used to form the DAE observer, with a feedback gain $K(x,z)$ introduced as

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

This observer estimates the unknown states x_1 and the algebraic variables x_2. Then, the residual equations in g_r from (4) 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 of Matlab's ODE solvers that are suitable for low-index DAE:s, e.g., `ode15s`, is generated using the class method `ObserverResGen`

```
sm.ObserverResGen( mso, 'ResGen' );
```

This call will generate the file `ResGen.m`, see Appendix B.4 for an example from the use case in Section 8.

Let an extended state vector be $w = (\hat{x}_1, \hat{x}_2, r)$, and the dimensions for $\hat{x}_1$, $\hat{x}_2$, 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

$$M\dot{w} = \begin{pmatrix} g_1(\hat{x}_1, \hat{x}_2, z) + K(\hat{x}, z) g_r(\hat{x}_1, \hat{x}_2, z) \\ g_2(\hat{x}_1, \hat{x}_2, z) \\ r - g_r(x_1, x_2, z) \end{pmatrix} = F(w, z) \tag{5}$$

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 `ode15s` 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:

```
M = [eye(2) zeros (2,4); zeros (4,6)];
[~,w] = ode15s(@(ts,x) ResGen( x, interp1(t,z,ts ),  K, params ),  ...
  t, x0, odeset('Mass',M));
r=w(:,6)
```

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

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

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

Let the estimation error be $e = x_1 - \hat{x}_1$, 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

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

See further details in the use case in Section 8.4 for examples on how to form `linpoint` and `parameters`, and also how to compute a locally stabilizing feedback gain K.

8 USE CASE

This section shows a simple use case how the toolbox can be used. As an example model, the three-tank model from [4] is used. All code is available, and possible to run, in file `usecase.m` found in the `examples/ThreeTankSimulation/` directory. The three-tank system is shown in Figure 8 and simple model of

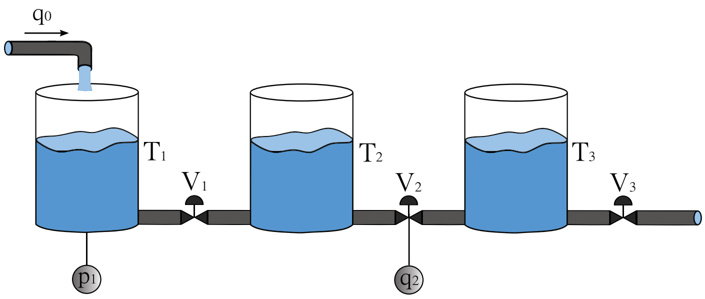


Figure 8: Diagram of the three-tank system.

the system is

$$
\begin{aligned}
e_1 &: q_1 = \frac{1}{R_{V1}}(p_1 - p_2) & e_7 &: y_1 = p_1 \\
e_2 &: q_2 = \frac{1}{R_{V2}}(p_2 - p_3) & e_8 &: y_2 = q_2 \\
e_3 &: q_3 = \frac{1}{R_{V3}}(p_3) & e_9 &: y_3 = q_0 \\
e_4 &: \dot{p}_1 = \frac{1}{C_{T1}}(q_0 - q_1) & e_{10} &: \dot{p}_1 = \frac{dp_1}{dt} \\
e_5 &: \dot{p}_2 = \frac{1}{C_{T2}}(q_1 - q_2) & e_{11} &: \dot{p}_2 = \frac{dp_2}{dt} \\
e_6 &: \dot{p}_3 = \frac{1}{C_{T3}}(q_2 - q_3) & e_{12} &: \dot{p}_3 = \frac{dp_3}{dt}
\end{aligned}
$$

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}.

8.1 Model definition

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

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

syms(model.x{:})
syms(model.f{:})
syms(model.z{:})
syms(model.parameters{:})

model.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
  dp2==1/CT2*(q1-q2) + fT2, ... % e5
```

```
  dp3==1/CT3*(q2−q3) + fT3, ... % e6
  y1==p1, y2==q2, y3==q0,... % e7, e8, e9
  DiffConstraint('dp1','p1 '),...  % e10
  DiffConstraint('dp2','p2 '),...  % e11
  DiffConstraint('dp3','p3 '),...  % e12
};

sm = DiagnosisModel( model );
sm.name = 'Three tank system';
```

With a model object, the model structure can be plotted using the simple command

```
sm.PlotModel();
```

which will produce Figure 9.

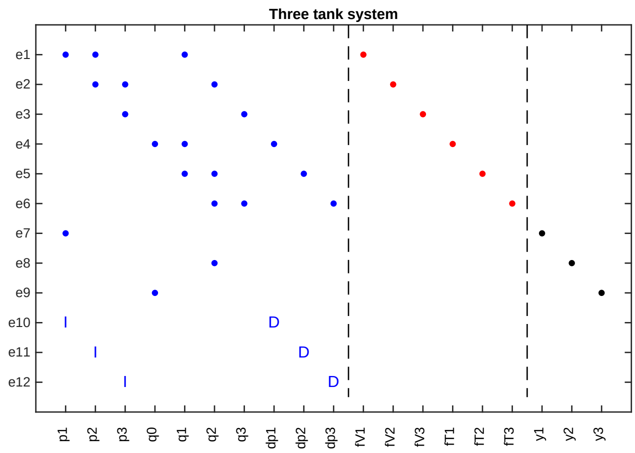


Figure 9: Three tank model structure.

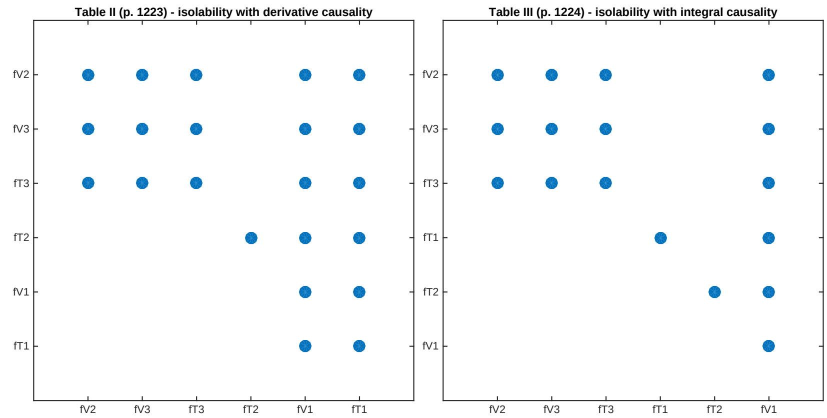
8.2 Isolability analysis

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 [4], 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

```
sm.IsolabilityAnalysis(' causality ',' der ');
sm.IsolabilityAnalysis(' causality ',' int ');
```

produces Figures 10-a and b. See [4] 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 $fV1$, $fT1$, and $fT2$, and 2) the group of faults $\{fV2, fV3, fT3\}$ 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

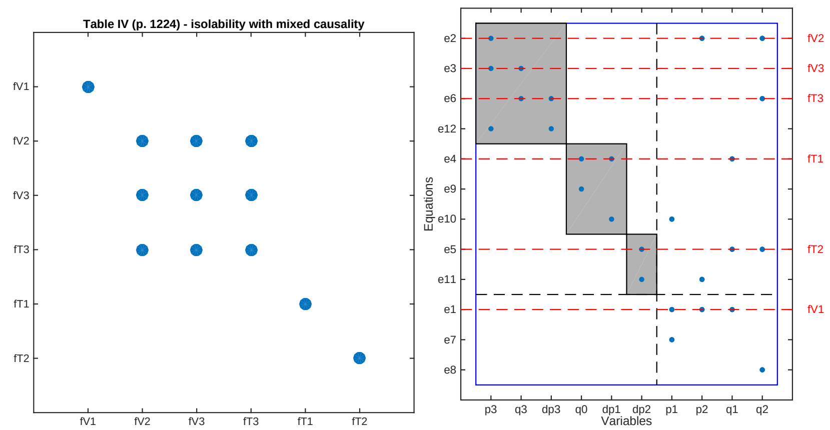


(a) Derivative causality **(b)** Integral causality

Figure 10: Isolability matrices in derivative and integral causality.

model only consist of an overdetermined part. The following commands in Matlab produces the figures

```
sm.IsolabilityAnalysis ();
sm.PlotDM('eqclass', true, 'fault', true );
```



(a) Mixed causality isolability matrix **(b)** Dulmage-Mendelsohn

Figure 11: Isolability analysis in the mixed causality case, together with the Dulmage-Mendelsohn decomposition. The figure shows the canonical decomposition of the overdetermined part, with indications where the faults appear in the model.

With the decomposition, the isolability properties of mixed causality case is clearly visible since the faults appear in different equivalence classes, except for the group $\{fV2, fV3, fT3\}$ which appears in the same class.

8.3 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 MSOs, i.e., the set of minimally structurally overde-

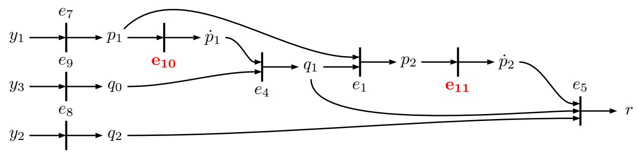


Figure 12: Residual r_1, sequential residual generator in derivative causality based on MSO $\mathcal{M}_2$.

termined sets of equations, see [8] for full details on how this is done. The Matlab command

```
msos = sm.MSO();
```

gives the following 6 MSOs

$$
\begin{aligned}
\mathcal{M}_1 &= \{e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\} \\
\mathcal{M}_2 &= \{e_1, e_4, e_5, e_7, e_8, e_9, e_{10}, e_{11}\} \\
\mathcal{M}_3 &= \{e_1, e_2, e_3, e_5, e_6, e_7, e_8, e_{12}, e_{11}\} \\
\mathcal{M}_4 &= \{e_1, e_2, e_3, e_4, e_6, e_7, e_8, e_9, e_{10}, e_{12}\} \\
\mathcal{M}_5 &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_8, e_9, e_{10}, e_{11}, e_{12}\} \\
\mathcal{M}_6 &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_9, e_{10}, e_{11}, e_{12}\}
\end{aligned}
\tag{6}
$$

This means that these 6 are the *minimal* sets of equations that has redundancy and therefore can be used to design residual generators.

8.4 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 (r_1), one in integral causality (r_2), and one in mixed causality (r_3). The forth residual generator (r_4) will be designed using a simple observer based approach. Residual generators r_1, r_2 and r_4 will use MSO $\mathcal{M}_2$ in (6) and r_3 will use $\mathcal{M}_1$.

The first residual generator is corresponds to Fig. 2 in [4], that use equation e_5 as a residual equation and the remaining, exactly determined, equations in $\mathcal{M}_2$ 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

```
Gamma1 = sm.Matching(setdiff(msos{2},5)); % compute matching
sm.SeqResGen( Gamma1, 5, 'ResGen1' );
```

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

If using e_7 instead of e_5 as a residual equation, with the same MSO, we obtain a residual generator in integral causality instead. The corresponding computational graph, same as Fig. 3 in [4], 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,

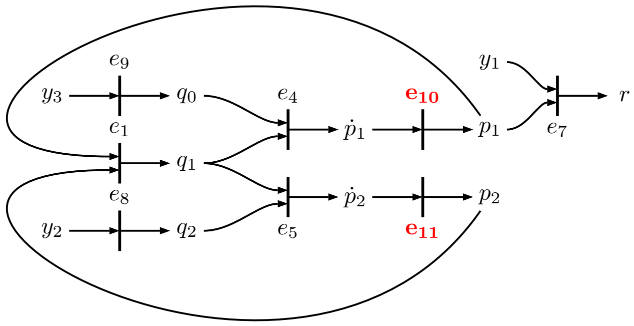


Figure 13: Residual r_2, sequential residual generator in integral causality based on mso $\mathcal{M}_2$.

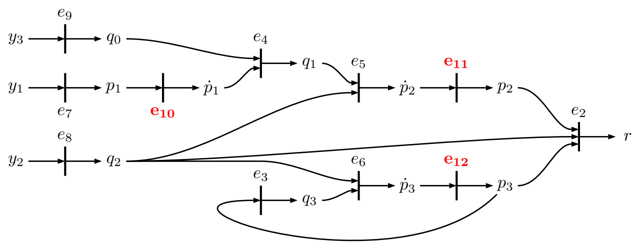


Figure 14: Residual r_3, sequential residual generator in mixed causality based on mso $\mathcal{M}_1$.

```
Gamma2 = sm.Matching(setdiff(msos{2},7)); % compute matching
sm.SeqResGen( Gamma2, 7, 'ResGen2' );
```

and the generated code is included in Appendix B.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 $\mathcal{M}_1$ with equation e_2 as residual equation. The computational graph is shown in Figure 14, which corresponds to Fig. 12 in [4]. Code generation is done again in the same way

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

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

In the fourth residual generator, mso $\mathcal{M}_2$ 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

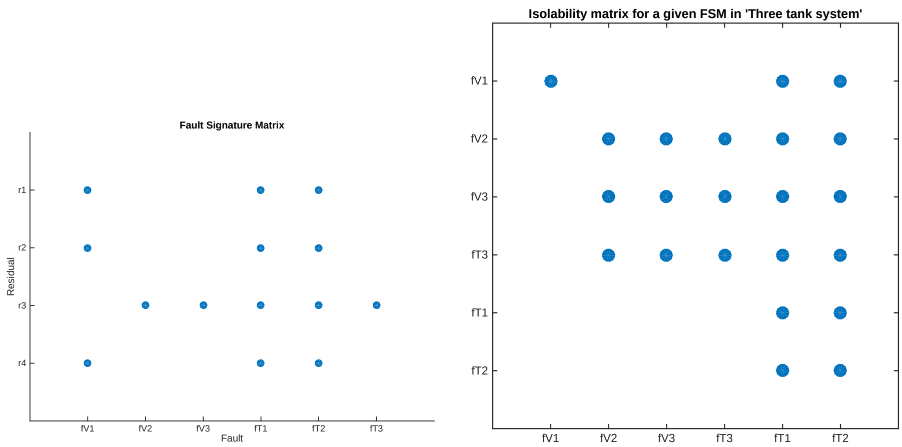
```
linpoint.x0 =  [0,0,0,0,0,0];   linpoint.z0 =  [0;0;0];
params.Rv1 = 1; params.Rv2 = 1; params.Rv3 = 1;
params.CT1 = 1; params.CT2 = 1; params.CT3 = 1;
[A,C] = sm.ObserverResGen( msos{2}, 'ResGen4', 'linpoint', linpoint ,...
  'parameters', params );
```

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

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

8.5 Isolability properties of residual generators

The isolability analysis in Section 8.2 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



(a) Fault signature matrix **(b)** Isolation performance

Figure 15: Fault signature matrix (FSM) and isolability properties for the four residuals $r_1, \ldots r_4$.

```
FSM = sm.FSM({msos{2}, msos{2}, msos{1}, msos{2}});
spy(FSM,30)
set(gca, 'YTick', 1:4, 'XTick', 1:sm.nf ,...
  'YTickLabel', {'r1 ', 'r2 ', 'r3 ', 'r4 '},' XTickLabel',sm.f ,...
  'box', 'off ');
xlabel ('Fault')
ylabel ('Residual')
 title ('Fault Signature Matrix')

sm.IsolabilityAnalysisFSM( FSM );
```

Figure 15-b should be compared to Figure 11-a and it is clear that the four residuals chosen does not reach the best possible isolation performance. This is not surprising since only MSOs $\mathcal{M}_1$ and $\mathcal{M}_2$ were used to generate the four residuals. If, for example residual generators for $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, and $\mathcal{M}_4$ were to be designed, full structural isolability would be achieved.

8.6 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

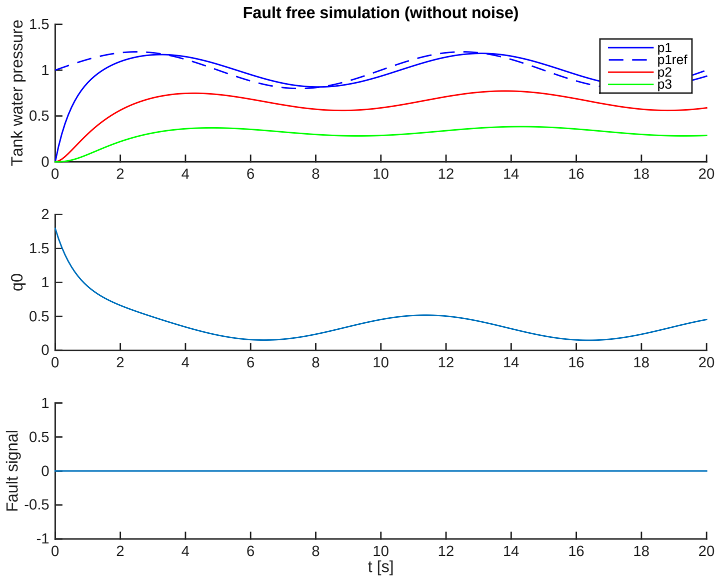


Figure 16: Simulation of fault free operation of the three-tank system.

Figure 17 where a ramp fault is added, starting at $t = 6$ and reaching top value at $t = 10$.

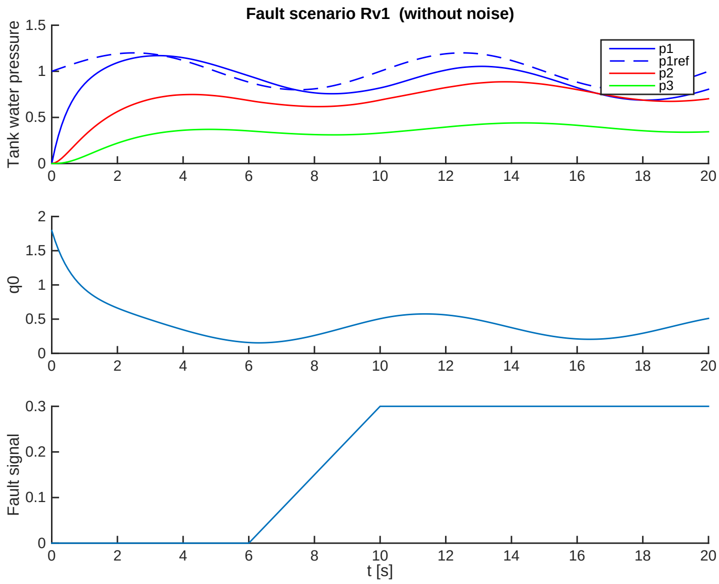


Figure 17: Simulation of the faulty case, f_{Rv1}, of the three-tank system.

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

```
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 (5) and note that a stiff, implicit, solver is needed to integrate the observer. In our case, we chose the solve `ode15s`. The code to integrate the residual generator is

```
M4 = [eye(2) zeros (2,4); zeros (4,6)];
[~,x] = ode15s(@(ts,x) ResGen4( x, interp1(t,z,ts ), K4, params ), ...
  t, x0, 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

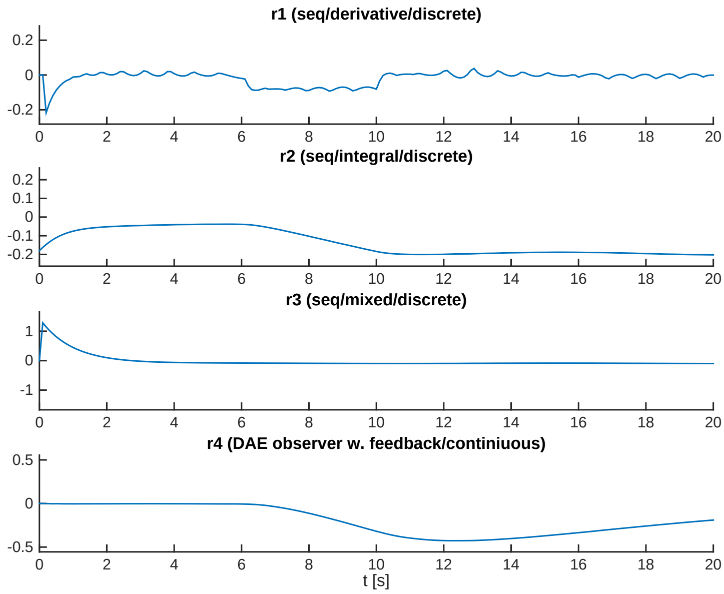


Figure 18: Residual values for residuals r_1, r_2, r_3, and r_4 for the fault mode f_{Rv1} shown in Figure 17.

clear that, after the initial transient all residuals react as expected. It is also clear that r_1 is more noise 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.

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
```

Method name	Description
Model object definition and manipulation	
DiagnosisModel	Constructor for model object used for diagnosis analysis and residual generator code generation
AddSensors	Add sensors to a model
LumpDynamics	Lump dynamic variables for structural model
Structural	Convert a symbolic model to a structural model
copy	Make a new copy of the model object
Model properties	
`x`	Unknown variables
`f`	Fault variables
`z`	Known variables
`X`	Incidence matrix for unknown variables
`F`	Incidence matrix for fault variables
`Z`	Incidence matrix for known variables
`e`	Equation names
`name`	Model name
`type`	Model type
`P`	List of possible sensor locations
`Pfault`	Which sensor locations may be faulty
`parameters`	List of model parameter names
`syme`	Symbolic equations
Model exploration	
`PlotDM`	Plots Dulmage-Mendelsohn decomposition of model structure
`PlotModel`	Plots a model object
`SubModel`	Extract submodel
`Lint`	Print model information and check for inconsistencies
`ne`	Number of equations in model
`nf`	Number of fault variables in model
`nx`	Number of unknown variables in model
`nz`	Number of known variables in model
`Redundancy`	Compute the structural degree of redundancy of a model

continued on next page

continued from previous page	
Method name	Description
`srank`	Compute the structural rank of the incidence matrix for the unknown variables
`IsHighIndex`	Is the model of high structural differential index?
`IsPSO`	Is the model proper structurally overdetermined
`BipartiteToLaTeX`	Generate a LaTeX document with the bipartite graph correponding to the structural model
Diagnosability analysis	
`DetectabilityAnalysis`	Performs a structural detectability analysis
`IsolabilityAnalysis`	Perform structural isolability analysis of model
`IsolabilityAnalysisArrs`	Perform structural isolability analysis of a set of ARRs
`IsolabilityAnalysisFSM`	Perform structural isolability analysis of a Fault Signature Matrix (FSM)
Sensor placement	
`PossibleSensorLocations`	Set possible sensor locations
`SensorLocationsWithFaults`	Set possible sensor locations that has faults in new sensors
`SensorPlacementDetectability`	Determine minimal set of sensors to achieve detectability
`SensorPlacementIsolability`	Determine minimal set of sensors to achieve maximal fault isolability
`CompiledMHS`	Use the compiled minimal-hitting set algorithm if available (not recommended)
Overdetermined equations	
`MSO`	Compute the set of MSOs
`MTES`	Computes the set of minimal test equation support
`FSM`	Compute the fault signature matrix (FSM)
`TestSelection`	A minimal hitting set based test selection approach
Residual generation	
`ObserverResGen`	Generate Matlab code for obsserver based residual generator
`Matching`	Compute a matching in the model for a set of equations
`SeqResGen`	Generate Matlab code for sequential residual generator
`MSOCausalitySweep`	For a given MSO, determine causality for sequential residual generator for each n residual equations

B GENERATED CODE IN USE-CASE

Below are the generated code for the four residual generators in the use case in Section 8.

B.1 ResGen1

```
function [r, state] = ResGen1(z,state,params,Ts)
% RESGEN1 Sequential residual generatorfor 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 11-Aug-2015 11:48:36

  % Parameters
  Rv1 = params.Rv1;
  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
  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
  if length(state.p1)==1
    state.p1 = p1;
  else
    state.p1 = [p1 state.p1(1)];
  end
  if length(state.p2)==1
    state.p2 = p2;
  else
    state.p2 = [p2 state.p2(1)];
  end
```

```
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
```

B.2 ResGen2

```
function [r, state] = ResGen2(z,state,params,Ts)
% RESGEN2 Sequential residual generatorfor 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: p1, p2

% File generated 11-Aug-2015 11:48:38

  % Parameters
  Rv1 = params.Rv1;
  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
  p1 = state.p1;
  p2 = state.p2;

  % Residual generator body
  q2 = y2; % e8
  q0 = y3; % e9
  q1 = (p1-p2)/Rv1; % e1
  dp2 = (q1-q2)/CT2; % e3
  dp1 = (q0-q1)/CT1; % e2
  p1 = ApproxInt(dp1,state.p1,Ts); % e4
  p2 = ApproxInt(dp2,state.p2,Ts); % e5
```

```
  r = -p1+y1; % e7

  % Update state variables
  if length(state.p1)==1
    state.p1 = p1;
  else
    state.p1 = [p1 state.p1(1)];
  end
  if length(state.p2)==1
    state.p2 = p2;
  else
    state.p2 = [p2 state.p2(1)];
  end
end

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

B.3 ResGen3

```
function [r, state] = ResGen3(z,state,params,Ts)
% RESGEN3 Sequential residual generatorfor 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, p1, p2

% File generated 11-Aug-2015 11:48:39

  % Parameters
  Rv1 = params.Rv1;
  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;
  p1 = state.p1;
  p2 = state.p2;
```

```
  % Residual generator body
  q2 = y2; % e8
  q0 = y3; % e9
  q3 = p3/Rv3; % e1
  dp3 = (q2-q3)/CT3; % e2
  p3 = ApproxInt(dp3,state.p3,Ts); % e3
  p1 = y1; % e7
  dp1 = ApproxDiff(p1,state.p1,Ts);  % e10
  q1 = q0-CT1*dp1; % e4
  dp2 = (q1-q2)/CT2; % e5
  p2 = ApproxInt(dp2,state.p2,Ts); % e11
  r = q2-(p2-p3)/Rv2; % e2

  % Update state variables
  if length(state.p3)==1
    state.p3 = p3;
  else
    state.p3 = [p3 state.p3(1)];
  end
  if length(state.p1)==1
    state.p1 = p1;
  else
    state.p1 = [p1 state.p1(1)];
  end
  if length(state.p2)==1
    state.p2 = p2;
  else
    state.p2 = [p2 state.p2(1)];
  end
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
```

B.4 ResGen4

```
function dx = ResGen4(x,z,K,params)
% RESGEN4 Observer based residual generatorfor 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 11-Aug-2015 11:48:40


  % Parameters
  Rv1 = params.Rv1;
  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);

  % 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
```

REFERENCES

[1] Mogens Blanke, Michel Kinnaert, Jan Lunze, and Marcel Staroswiecki. *Diagnosis and fault-tolerant control*. Springer, 3rd edition, 2016.

[2] Andrew L Dulmage and Nathan S Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10(4):516–534, 1958.

[3] Dilek Dustegör, Erik Frisk, Vincent Coquempot, Mattias Krysander, and Marcel Staroswiecki. Structural analysis of fault isolability in the DAMADICS benchmark. *Control Engineering Practice*, 14(6):597–608, 2006.

[4] Erik Frisk, Anibal Bregon, Jan Åslund, Mattias Krysander, Belarmino Pulido, and Gautam Biswas. Diagnosability analysis considering causal interpretations for differential constraints. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 42(5):1216–1229, 2012.

[5] C William Gear and Linda R Petzold. Ode methods for the solution of differential/algebraic systems. *SIAM Journal on Numerical Analysis*, 21(4):716–728, 1984.

[6] Mattias Krysander, Jan Åslund, and Erik Frisk. A structural algorithm for finding testable sub-models and multiple fault isolability analysis. 21st International Workshop on Principles of Diagnosis (DX-10), Portland, Oregon, USA, 2010.

[7] Mattias Krysander and Erik Frisk. Sensor placement for fault diagnosis. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 38(6):1398–1410, 2008.

[8] Mattias Krysander, Jan Åslund, and Mattias Nyberg. An efficient algorithm for finding minimal overconstrained subsystems for model-based diagnosis. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 38(1):197–206, 2008.

[9] J. Armengol Llobet, A. Bregon, T. Escobet, E. R. Gelso, M. Krysander, M. Nyberg, X. Olive, B. Pulido, and L. Trave-Massuyes. Minimal structurally overdetermined sets for residual generation: A comparison of alternative approaches. In *Proceedings of IFAC Safeprocess'09*, Barcelona, Spain, 2009.

[10] Linda Petzold. Differential/algebraic equations are not ode's. *SIAM Journal on Scientific and Statistical Computing*, 3(3):367–384, 1982.

[11] Albert Rosich, Erik Frisk, Jan Åslund, Ramon Sarrate, and Fatiha Nejjari. Fault diagnosis based on causal computations. *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans*, 42(2):371–381, 2012.

[12] Carl Svärd and Mattias Nyberg. Residual generators for fault diagnosis using computation sequences with mixed causality applied to automotive systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 40(6):1310–1328, 2010.

[13] Carl Svärd, Mattias Nyberg, and Erik Frisk. Realizability constrained selection of residual generators for fault diagnosis with an automotive engine application. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(6):1354–1369, 2013.

[14] Carl Svärd, Mattias Nyberg, Erik Frisk, and Mattias Krysander. Automotive engine FDI by application of an automated model-based and data-driven design methodology. *Control Engineering Practice*, 21(4):455–472, 2013.

C INDEX OF KEYWORDS AND METHODS

KEYWORDS

METHODS