SELF-RECONFIGURABLE SHIP FLUID-NETWORK MODELING FOR SIMULATION-BASED DESIGN

A Thesis
Presented to
The Academic Faculty

by

Kyungjin Moon

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Aerospace Engineering

Georgia Institute of Technology
August 2010

Copyright © 2010 by Kyungjin Moon

SELF-RECONFIGURABLE SHIP FLUID-NETWORK MODELING FOR SIMULATION-BASED DESIGN

Approved by:

Professor Dimitri N. Mavris,
Committee Chair
School of Aerospace Engineering
Georgia Institute of Technology

Professor Daniel P. Schrage
School of Aerospace Engineering
Georgia Institute of Technology

Professor Mark Costello
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Neil Weston
School of Aerospace Engineering
Georgia Institute of Technology

Mr. Frank Ferrese
Naval Surface Warfare Center
Carderock Division

Date Approved: May 19, 2010

To my love, Minsuk,

my son,

and

my parents.

ACKNOWLEDGEMENTS

During my entire time in the Ph.D program, I have felt like I was a novice marathoner, who was running his first match. As with all other rookies in their first journeys, I would have never been able to make it to this successful finale of mine if I were not with a great coach and people who supported me. I would first like to thank my advisor, Professor Dimitri Mavris. He gave me, in the perfect balance, both freedom of exploring various research directions – even some crazy ideas – and thoughtful and insightful advice, monitoring, and redirection of my research as a great scholar, sincere mentor, and solid supporter of mine.

I would also like to thank Dr. Neil Weston, one of my committee members, for giving advice, reviewing my work, and sometimes helping this struggling "international" graduate student with his technical writing. I want to thank the rest of my thesis committee, Professor Daniel Schrage, Professor Mark Costello, and Mr. Frank Ferrese, for their constructive suggestions and feedback on my thesis work.

I would also like to thank the former and current members of IRIS project team: Michael Balchanos, David Fullmer, Matt Hoepfer, Joosung Kang, Dr. Yongchang Li, Bassem Nairouz, and Daili Zhang. It has been pure pleasure having discussions with you folks in the many meetings and road trips and working with you all for about 6 years. All these are a part of my invaluable memories.

I want to thank my wife, Minsuk. During 7 years of graduate studies, you always gave me cheers and patience on the numerous days of my coming home late in the night with a stressful look on my face. Also, thank you for taking good care of our most valuable treasure, our son.

Thank you, mother and father. I could never thank you two enough for all

your sacrifices through your lives. You always have been my motivation, inspiration, courage, and shelter of my life, and you will be so forever.

Thank you all,

Kyungjin, Spring 2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS OR ABBREVIATIONS

$\delta(G)$	Minimal Degree of the Vertices of Graph G.
$\Delta(G)$	Maximal Degree of the Vertices of Graph G.
$d(x)$	Degree of Vertex x (number of adjacent vertices to x).
E	Edge Set of a Graph.
$E(G)$	Edge Set of Graph G.
G	A Graph.
V	Node or Vertex Set of a Graph.
$V(G)$	Node or Vertex Set of Graph G.
$\|G\|$	Order of Graph G (=$\|V\|$, number of vertices).
$\|X\|$	Number of elements in Set X.
$\|\|G\|\|$	Size of Graph G (=$\|E\|$, number of edges).
AAW	Anti-Air Warfare.
AFRL	Air Force Research Laboratory.
ARIMA	Auto-Regressive Integrated Moving Average (model).
ARMA	Auto-Regressive Moving Average (model).
ARMAX	Auto-Regressive Moving Average with eXogenous input (model).
ARX	Auto-Regressive with eXogenous inputs (model).
BIBO	Bounded-Input, Bounded-Output.
BJ	Box-Jenkins (model).
BPTT	Backpropagation through Time.
DACE	Design and Analysis of Computer Experiments.
DAE	Differential Algebraic Equations.
DD(X)	Next-Generation Naval Surface Combatants Program.
DOE	Design of Experiment.
EOA	Energy Optimized Aircraft.

ERLS	Extended Recursive Least-Squares.
FIR	Finite Impulse Response.
IED	Integrated Electric Drive.
IEP	Integrated Engineering Plant.
INVENT	Integrated Vehichle & Technology program.
IPS	Integrated Power System.
LCC	Life Cycle Cost.
M&S	Modeling and Simulation.
MA	Moving-Average (model).
MLE	Maximum Likelihood Estimation.
MLP	Mult-Layer Perceptron.
MSE	Mean Squared Error.
NN	(Artificial) Neural Network.
NRAC	Naval Research Advisory Committee.
O&S	Operation and Support.
OCR	Operation Capability Rate.
OE	Output-Error (model).
ONR	Office of Naval Research.
RBF	Radial-Basis Function.
RNN	Recurrent Neural Network.
RSM	Response Surface Method.
RTRL	Real-Time Recurrent Learning.
SHL	Single Hidden Layer.
T&E	Test & Evaluation.

SUMMARY

Our world is filled with large-scale engineering systems, which provide various services and conveniences in our daily life. A distinctive trend in the development of today's large-scale engineering systems is the extensive and aggressive adoption of automation and autonomy that enable the significant improvement of systems' robustness, efficiency, and performance, with considerably reduced manning and maintenance costs, and the U.S. Navy's DD(X), the next-generation destroyer program, is considered as an extreme example of such a trend.

This thesis pursues a modeling solution for performing simulation-based analysis in the conceptual or preliminary design stage of an intelligent, self-reconfigurable ship fluid system, which is one of the concepts of DD(X) engineering plant development. Through the investigations on the Navy's approach for designing a more survivable ship system, it is found that the current naval simulation-based analysis environment is limited by the capability gaps in damage modeling, dynamic model reconfiguration, and simulation speed of the domain specific models, especially fluid network models.

As enablers of filling these gaps, two essential elements were identified in the formulation of the modeling method. The first one is the graph-based topological modeling method, which will be employed for rapid model reconstruction and damage modeling, and the second one is the recurrent neural network-based, component-level surrogate modeling method, which will be used to improve the affordability and efficiency of the modeling and simulation (M&S) computations. The integration of the two methods can deliver computationally efficient, flexible, and automation-friendly M&S which will create an environment for more rigorous damage analysis and exploration of design alternatives.

As a demonstration for evaluating the developed method, a simulation model of a notional ship fluid system was created, and a damage analysis was performed. Next, the models representing different design configurations of the fluid system were created, and damage analyses were performed with them in order to find an optimal design configuration for system survivability. Finally, the benefits and drawbacks of the developed method were discussed based on the result of the demonstration.

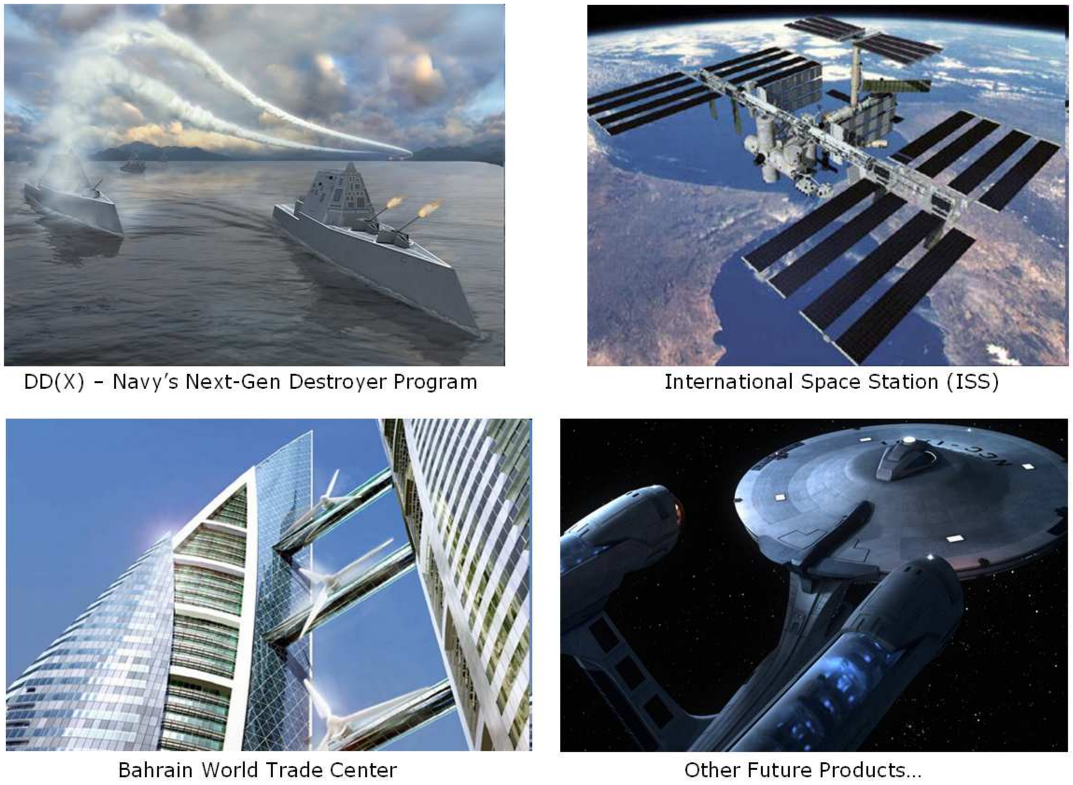
CHAPTER I

INTRODUCTION

During recent decades, engineering systems around us have evolved to be more and more integrated into larger systems in a way to provide better quality of functions, services, and conveniences. Such products can be found everywhere in our life – such as automobiles, aircraft, ships, and buildings – and have a commonality in their structures, which are composed of a number of networks representing different functional and physical domains, such as mechanical, fluid, electrical, communication, and control systems, just to name a few.

One distinctive trend in the development of the large-scale engineering systems is the extensive and aggressive adoption of automation and autonomy that enable the significant improvement of systems' robustness to failure, energy efficiency, service performance, and functional bandwidth, with considerably reduced manning and maintenance costs. However, this approach increases the system complexity, which causes the high cost of development, technical difficulties in predicting its behaviors, and understanding interactions among the subsystems. And evidently, theses difficulties make the design and development of a large-scale complex system one of new and imminent challenging problems in today's engineering and scientific communities.

Meanwhile, with advances in computer technologies and mathematical algorithms, computer modeling and simulation (M&S) has achieved enormous success and prosperity in the past several decades, aiding various engineering and scientific breakthroughs. These days, M&S is often a key process of test and evaluation (T&E) in the research, development, and design processes, since it provides a more affordable and flexible way of analyzing and predicting engineering systems than those with



DD(X) – Navy's Next-Gen Destroyer Program

International Space Station (ISS)

Bahrain World Trade Center

Other Future Products...

Figure 1: Examples of Complex Systems in Engineering Domain

physical test and prototyping, which are very expensive to build and highly limited regarding model modifications by design change.

Observing such benefits and the fast progresses of the field of computer simulation, many engineers and scientists envision simulation-based engineering and design as the enabler of developing many large-scale engineering systems, but its potential is not yet fully unleashed because of various problems and limitations that have not been solved until now. According to the report by National Science Foundation [56, p.25], the key challenges in M&S of complex systems are that: 1) the development of models is highly time-consuming and requires advanced technical skills and knowledge; 2) linking multi-scale, multi-physics models is still a widely unsolved problem; and 3) since simulation is treated and performed as a separate discipline from design process, the collaboration with a design optimization process is often highly limited.

These problems become more aggravated especially in an early design stage of complex systems, since there are only limited human, financial, and temporal resources available in this stage, resulting that a M&S-based analysis approach does not enter into the design cycle until its later stages [56, p.23]. Therefore, the M&S-based design efforts may remain in each domain specific, functional-system level, without the integration for analyzing the aggregated system.

The goal of this thesis is to develop a modeling method and an M&S environment that can address some of the previously stated problems in the early design stages of a large-scale engineering system. The developed M&S method does not mean to serve as a general solution for any complex engineering system, because the application of M&S is very problem-specific, and requires different strategies and approaches based on the physical domain, assumptions, desired fidelity, and scope of M&S. In this thesis, the application domain is chosen to be a next-generation naval platform, especially the fluid system layer of it, and first of all, the further descriptions of the application domain and the motivation follow in the next sections.

1.1 Overview of DD(X), Next-Generation Naval Surface Combatants Program

The Navy's DD(X) program was established in order of the research and development of the next-generation, multi-mission naval destroyer [27], which would provide incomparable improvements of combat effectiveness and survivability, while sufficiently reducing total ownership cost. In order to achieve better combat effectiveness and survivability, a number of revolutionary technologies were incorporated in the DD(X) program, such as advanced gun system, integrated systems and computing environment, wave-piercing tumble-home hull, and stealthy body, just to name a few (see Table 1 for details). This program was started and conceptualized from DD-21 program, the predecessor of DD(X) program, that lasted from late 1990s to 2001. Then, the U.S. congress approved to perform detailed design and procure the lead ship in

Figure 2: Artwork of DDG-1000 Zumwalt Class (from www.navsource.org [55])

2005, and the Navy designated, in 2006, the ship's hull number 1000, and named this new destroyer after Zumwalt, the Chief of Naval Operations from 1970 to 1974 [58]. Figure 2 shows the artwork of the DDG-1000 class destroyer. The construction of the lead ship was scheduled in July 2008, but the procurement plan was down-sized to the delivery of only two ships from the Navy's original plan of seven ships, because of the pressure of the excessive increase of the program cost. Currently, the both ships are currently being built by Northrop Grumman Ship Systems and General Dynamics Bath Iron Works, and scheduled to be delivered in 2012.

1.1.1 Concepts of DD(X)

The design of engineering systems in DD(X) is based on two revolutionary concepts, which are Integrated Power System (IPS) [19] (see Figure 4) and Integrated Engineering Plant (IEP) [21, 39, 81]. The concept of IPS, which was initiated in 1994, can be summarized as an "all-electric ship." In the conventional ship-power system architecture, which is shown in Figure 3(a), there are two totally isolated sets of power

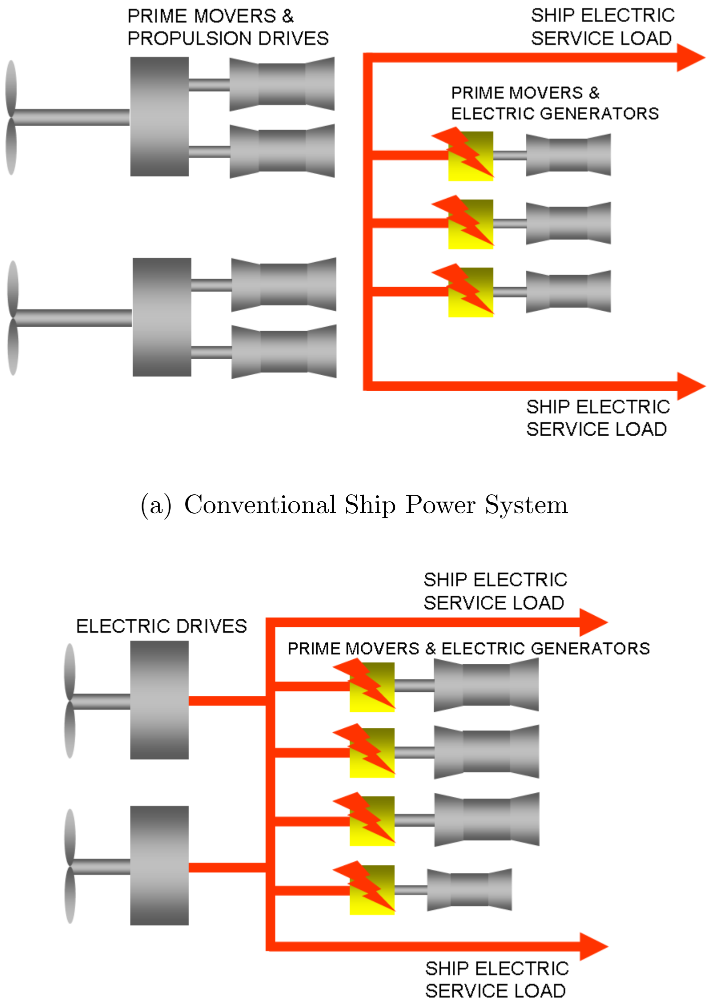
Table 1: Engineering Development Model (EDM) of DDG-1000 Acquisition [76]

Engineering Development Model	Description
Advanced Gun System [10]	Will provide long-range fire support for forces ashore through the use of unmanned operations and the long-range land attack projectile.
Autonomic Fire Suppression System [43]	Intended to reduce crew size by providing a fully automated response to fires.
Dual Band Radar	Horizon and volume search improved for performance in adverse environments.
Hull Form	Designed to significantly reduce radar cross section.
Infrared Mock-Up	Seeks to reduce ships heat signature in multiple areas.
Integrated Deckhouse and Aperture	Composite structure that integrates apertures of radar and communication systems.
Integrated Power System	Power system that integrates power generation, propulsion, and power distribution and management.
Integrated Undersea Warfare System	System for mine avoidance and submarine warfare with automated software to reduce workload.
Peripheral Vertical Launch System	Multi-purpose missile launch system located on the periphery of the ship to reduce damage to ship systems.
Total Ship Computing Environment [54]	Provides single computing environment for all ship systems to speed up command while reducing manning.

generation units (e.g., turbine engines) dedicated separately to the ship's mechanical propulsion and the rest of the loads in the ship system, such as weapons systems and auxiliary electric loads. In IPS, as shown in Figure 3(b), the ship's power generation and management are integrated by replacing the mechanical propulsion units with electric motors, so the single set of electric power generation units provides power for both the propulsion and all other electric loads in the ship. The power system is designed to be highly modularized so that the entire ship power system can be designed and manufactured in a manner of plug-and-play. According to Doerry et al. [19], the integrated electric drives (IED) and the modularized power system architecture of IPS could provide far better efficiency and flexibility in ship designing and manufacturing, and greater energy efficiency by intelligent utilization of excess power from power generation units. It could also improve maintainability and upgrade capability of the ship, leading to the reduced life cycle cost (LCC).

IEP is a more aggressive extension of IPS concept. Taking advantage of the IPS architecture, IEP pursues utilizing sensors and electric actuators throughout the entire ship system. This concept is aiming at a leap-ahead shift of the paradigm in ship's engineering operations – from human-based operations to autonomous or automation-based operations by the well-designed, intelligent layers of control systems.

The IEP concept was promoted for two main objectives, and the first one is to reduce the manning for ship operations. According to a report from Naval Research Advisory Committee (NRAC) in 2000 [74], the Navy's total budget had been decreased since 1985 by about 40%, but Operation and Support (O&S) costs had remained almost unchanged. Since the manning cost was over 50% of the O&S cost, it was unavoidable for the Navy to pay a lot of efforts for reducing manning in order to decrease O&S cost. The Navy's effort to reduce manning is traced back to DD-21 program that set a very ambitious goal of a 95 man crew, meaning 70% less O&S cost than a DDG-51 class ship, which needs a 350 man crew. The idea was the development of a real-time monitoring environment of the system conditions and failures for the crews in a remote area, using a dense network of sensors weaved with a wireless-networking technology and data fusion technologies [69]. Later in the DD(X) program, the goal was changed to a less aggressive 120 to 140 man crew, with the added capabilities of automated reconfiguration and damage control of engineering systems. In a conventional warship, most of the reconfiguration works and the damage control activities are performed manually.



(a) Conventional Ship Power System

(b) Integrated Power System

Figure 3: Comparison of Conventional Ship Power System and Integrate Power System

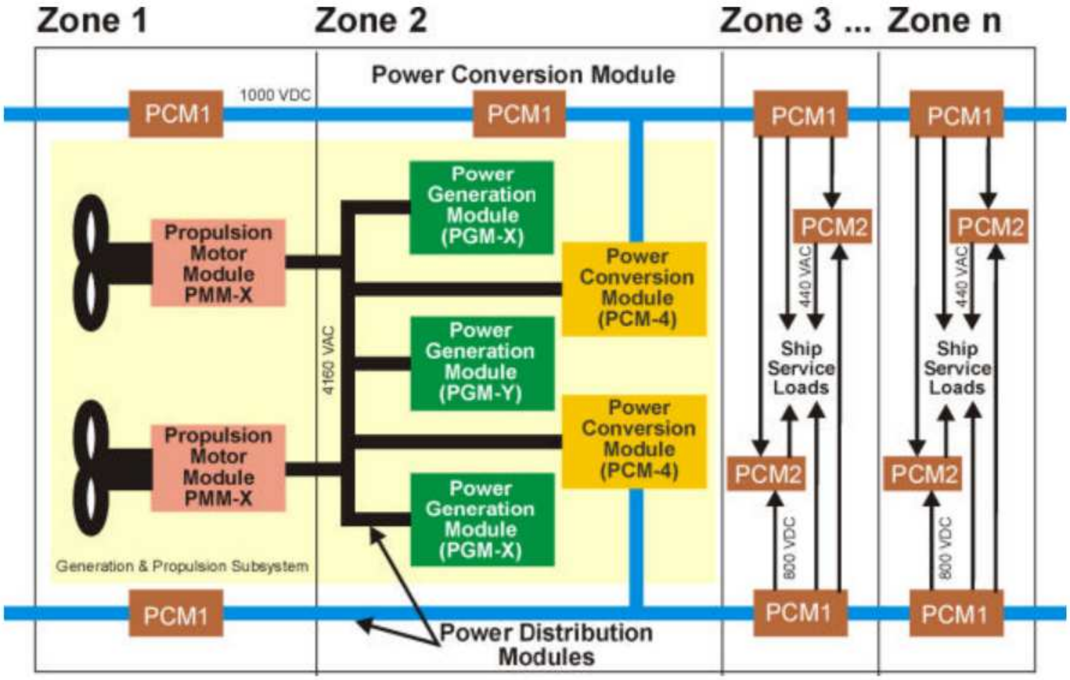

Zone 1
Zone 2
Zone 3 ... Zone n
Power Conversion Module
1000 VDC
PCM1
PCM1
PCM1
PCM1
Power Generation Module (PGM-X)
Propulsion Motor Module PMM-X
Power Conversion Module (PCM-4)
PCM2
PCM2
440 VAC
440 VAC
Ship Service Loads
Ship Service Loads
4160 VAC
Power Generation Module (PGM-Y)
Propulsion Motor Module PMM-X
Power Conversion Module (PCM-4)
PCM2
PCM2
800 VDC
800 VDC
Power Generation Module (PGM-X)
Generation & Propulsion Subsystem
PCM1
Power Distribution Modules
PCM1
PCM1

Figure 4: Notional Layout of Integrated Power System (from Doerry et al. [19])

Another objective of IEP is to ensure far better survivability of the ship under various damage conditions. This type of survivability is often expressed in such phrases as graceful degradation, fight-through capability, or puncture-proof capability. For a conventional naval warship, many of its subsystems (e.g., electric power , cooling , and fire-main systems) typically rely on manual operations of human experts for system recovery, when it comes to damage or malfunctions in subsystems, and the previous incidents clearly showed the high vulnerability of the human-based damage control. In 1987, USS Stark (FFG 31) was struck by two Exocet missiles launched by an Iraqi Mirage figherjet (see Figure 6(a)). The first missile was luckily misfired, but with the detonation of the second one, the consequence became deadly. It took about 50 minutes until the damage control teams partially restored the ruptured fire-main, while the fire spread out rapidly. After about another one hour, all engines had to be shut off. The fire lasted for about 12 hours incinerating the radar room and combat information center, claiming 37 casualties [5, 3]. The current Navy state-of-the-art,

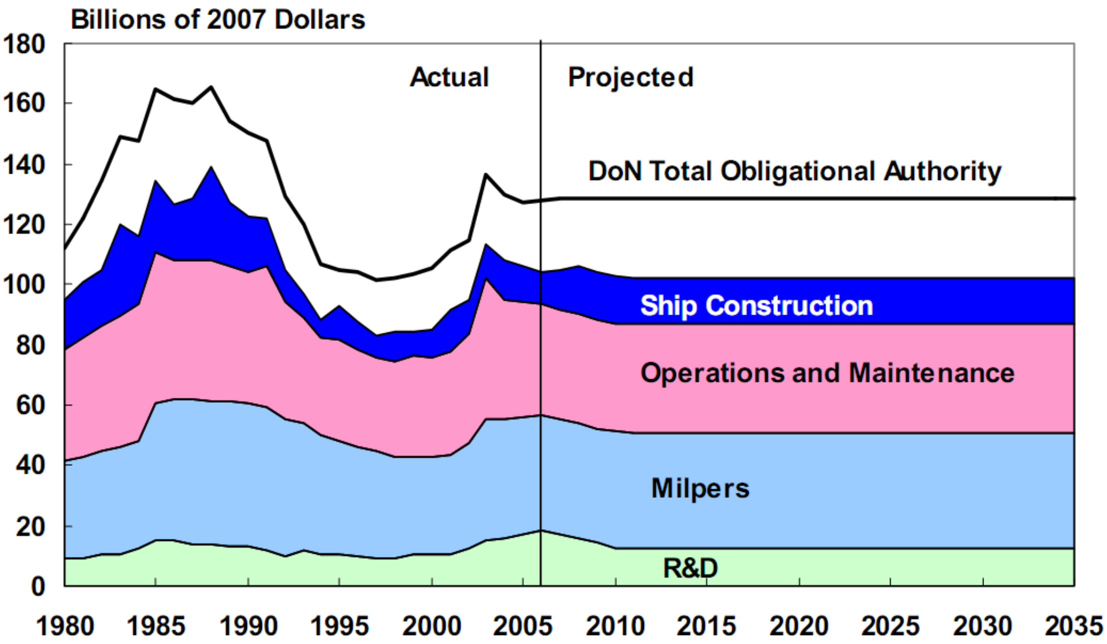
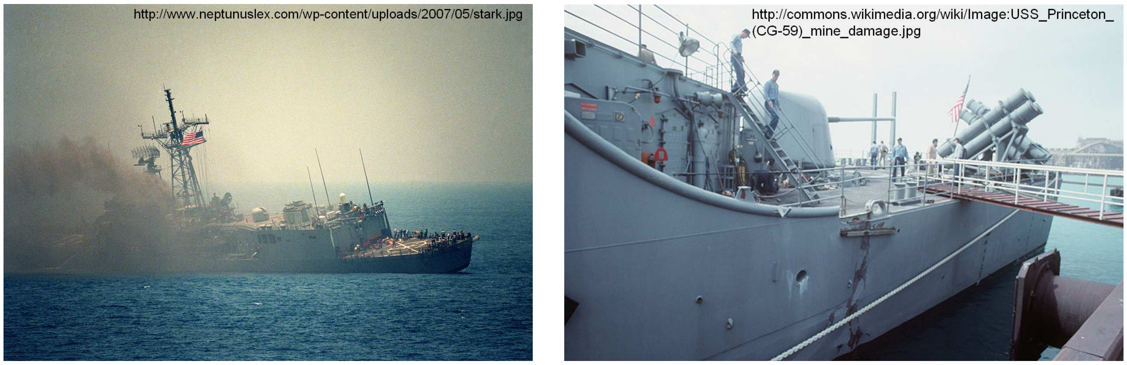


Figure 5: Navy Budget Chart [88]

Aleigh Burke (DDG 51) and Ticonderoga (CG 47) classes were no better than that older and smaller frigate in the way of damage control. In 1991, USS Princeton (CG-59) was damaged by the explosion of two mines (see Figure 6(b)), but its Aegis anti-air warfare (AAW) system could be back in operation within 2 hours, by the great works of its damage control teams [4]. Although the case of USS Princeton was considered as one of the successful stories in the Navy's record of damage control (in



http://www.neptunuslex.com/wp-content/uploads/2007/05/stark.jpg

http://commons.wikimedia.org/wiki/Image:USS_Princeton_(CG-59)_mine_damage.jpg

(a) USS Stark (FFG-31) Struck by Two Exocet Missiles (1987)

(b) USS Princeton (CG-50) with Mine Damage (1991)

Figure 6: Damage Examples of Conventional Warships

fact, the damage of USS Princeton was not so serious as that of USS Stark either), the fate of both USS Stark and USS Princeton would not be much different if there were successive hostile actions by the enemy. IEP is being designed to prevent the system from cascading catastrophes, and maximize the survivability to damage of the ship. Unlike the human-based damage controls, of which the response time is mostly in an order of several hours and no faster than several minutes, the autonomous controls can provide reflex actions of isolating time-critical failures such as the ruptures or damages of fuel pipes, electric power lines, cooling systems, and fire-main, and ultimately, reconfigure subsystems to provide various resources continuously to the vital systems of the ship within seconds or almost instantaneously.

1.1.2 IEP Control Architecture [21]

Apparently, the control architecture and its algorithms are the core of IEP concept. In the naval research for DD(X) engineering systems control, the agreed solution is a hierarchical, distributed control architecture. The IEP control architecture is basically constructed of three hierarchical layers, as shown in Figure 7 [21, 81].

The component layer in Figure 7 refers to the group of module/component-level controllers (or agents) that are embedded in various components of the ship's hydraulic, mechanical, and electrical systems. These controllers have direct interfaces with the sensors and actuators of their components, and regulates or optimizes the status of the components based on their local objectives. With relatively low-level logics and inferencing algorithms, they perform first-aid-type, reactive actions to time critical failures like ruptures, power disconnections, or fire in the ship systems.

The process layer in Figure 7 performs the control and coordination of the low-layer components to achieve the optimized availability of system-level functions such as propulsion, power generation and distribution, cooling, and damage control. Each process agent is capable of monitoring, diagnosing, and reconfiguring the engineering

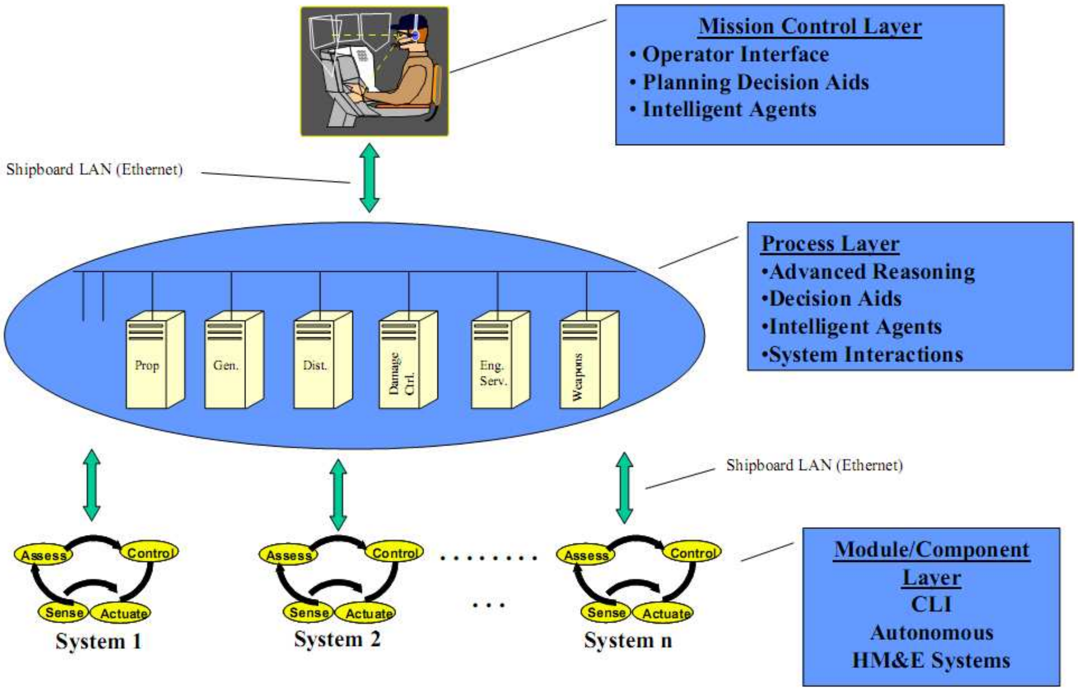


Figure 7: IEP Control Architecture [21]

systems by communicating with the associated component-level agents. To do these, it must have a high-level of intelligence and inferencing abilities, which can generate robust and optimized decision-making in order to secure system-level functionalities.

Lastly, the mission control layer performs a top-level decision-making by interacting with human operators. In this layer, the very abstract, high-level operational conditions such as dockside, normal cruise, combat, and damage control of the ship is decided, and translated into the system-level functional requirements and objectives for the control agents in the process layer. It also performs an intelligent planning of ship resource management based on the assigned mission, operating scenarios, and availability and demands of resources. During its decision-making process, the operator intervention is allowed under advisory information and warnings provided by the control system.

1.2 *Integrated M&S of Ship Engineering Systems for Fail-Proof Design*

As previously stated, one of the goals of the Navy's DD(X) program for the next-generation destroyer is to reduce manning levels to less than half of DDG-51 destroyers, while increasing ship survivability significantly. In order to meet the challenging requirement, Office of Naval Research (ONR) had conceptualized highly intelligent and distributed autonomy and automation in ship operations and damage controls for reconfiguration via smart actuators to recover from damages and malfunctions. Sucessful research progress includes the demonstration of agent-based control implemented into a chilled-water system testbed called Chilled Water Reduced Scale Advanced Demonstrator (CW-RSAD) built by Naval Surface Warfare Center, Carderock Division (NSWC-CD) [39, 66]. CW-RSAD is shown in Figure 8.

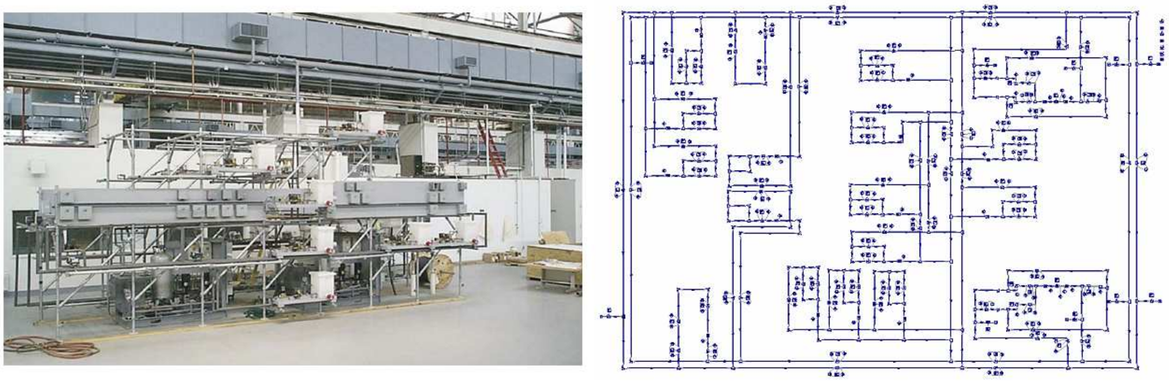


Figure 8: Physical Testbed [66] and Software (*FlowMaster2* ®) Model of CW-RSAD

As the next step, ONR is moving further into developing a multi-domain, multi-physics modeling and simulation (M&S) environment which will not be crucial only for designing control systems, but also for designing any engineering system in a ship. This M&S environment should be based on a system-of-systems perspective rather than unrealistic single-system responses. As a part of this effort, the Navy has developed a model integration environment called DOMINO [90]. The DOMINO environment uses the SQL server as the backbone of storing and exchanging all the simulation results of the models created by domain-specific M&S applications synchronously, so the simulation can capture the interactions and interdependencies between the ship subsystems when failures or damages occur in the simulation.

1.3 Problems of Domain-Specific Engineering Network Models in Integrated M&S

The Navy's multi-disciplinary M&S environment based on the integration of domain specific M&S tools is inevitable for simulation-based analyses in early design phases where very limited human, financial, and temporal resources are assigned, but it is easily challenged by three common problems of local domain tools. The following sections describe these three problems raised by domain-specific modeling tools.

1.3.1 Problem 1: Damage Modeling

The models created with domain-specific modeling tools are often incapable of, or poor at, performing damage simulation unless the modeling tools provide the libraries for modeling damages. The Navy's DOMINO environment is a good example for addressing this problem. The DOMINO model includes an electric power model, communication models, and fluid models. Among them, the fluid models were created with a 1-D pipeline simulation tool called *Flowmaster*®, which contains no library for damage or rupture analyses. As a rudimentary approach to modeling damage, a branched-off valve with one of its ends open to ambient pressure has been placed at a predefined location of interest in the fluid model [15] (see Figure 9). Along with its problem of incorrect modeling, this approach is unable to support a rigorous and extensive damage analysis which will be key to the design for resiliency and survivability, since it is virtually impossible to automate the damage analysis in a design process for a large set of damage scenarios.

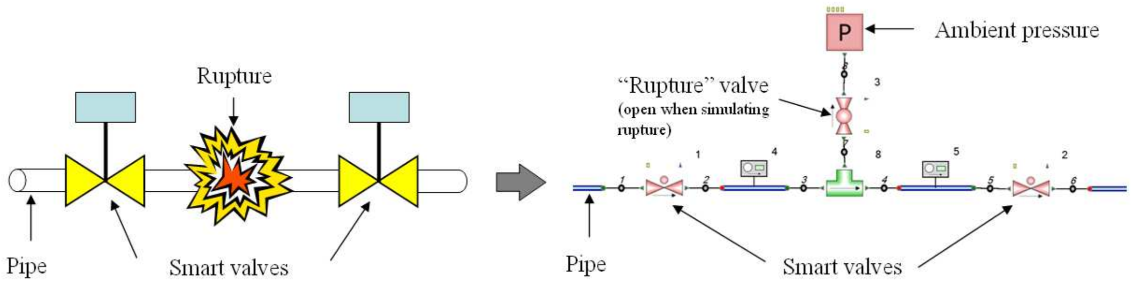


Figure 9: Example of Simple Ruptured Pipeline and its Implementation in Flowmaster

1.3.2 Problem 2: Model Reconfiguration

An early design phase is typified by the terms of aggressive design changes such as design space exploration, generation of various design alternative, and design optimization [49]. Recalling that many of the ship engineering systems are in the form of networks of their components, M&S based analyses should let systems engineers explore not only the component-level system configurations, but also the designs of different topological configurations among components, since the topology of the networked system affects both the performance and the cost of the entire system. As a result, the design of a large-scale system requires a very large number of design alternatives and simulation runs, meaning that an automation of M&S is a condition that must be achieved, not just preferred, for successful design of complex systems. However, in most commercial or legacy domain tools, a topological design change requires a manual modification of the baseline model. Figure 10 shows the example of a fluid model created by Flowmaster.

1.3.3 Problem 3: Simulation Cost

The dependence on domain specific M&S tools for running simulation often slows down the overall simulation speed and creates a large computational burden when it comes to simulation-based design approach. In the Navy DOMINO environment, an example is, again, the fluid models created using Flowmaster, which were slower than

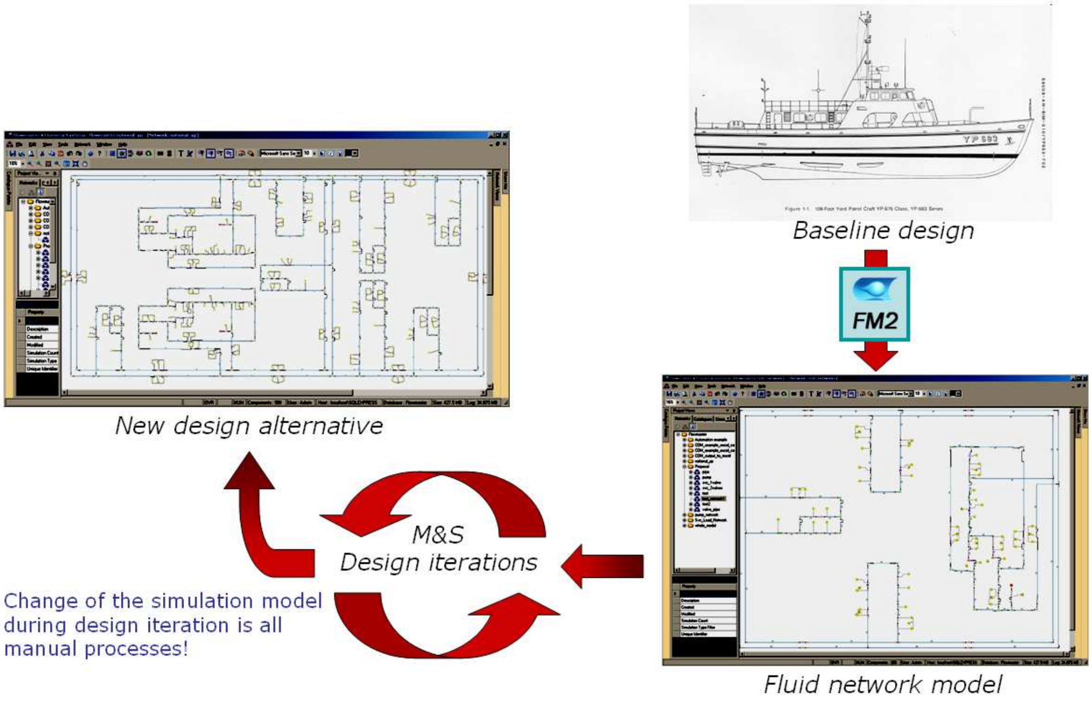


Figure 10: Example of Configuration Design Changes

the rest of the models – so slow that their simulation speed dominated the speed of the integrated simulation. The reason for their computational cost was not only in the simulation cost of this tool, but also the computational overhead or inefficiency from its external interfacing to the integrating framework. This high computational cost can prohibit many design-oriented analyses that require a large number of simulation cases.

1.4 Available M&S Methods

There are several M&S methods that are frequently applied to the analyses of various types of complex systems. Each of them has its own strengths, but it alone is not a feasible solution for addressing all the three problems discussed in §1.3. In this section, each of them are briefly introduced, and their strengths and weaknesses are discussed based on the previously discussed problem for the M&S of large-scale engineering networks.

1.4.1 Surrogate Modeling

As mentioned in the Navy M&S application, the cost of computer simulations is often a great obstacle for proceeding with various analyses for design or optimization, especially when it comes to complex systems. In order to achieve the successful design of a complex system, a large number of simulations need to be performed in order to evaluate mission effectiveness, solution robustness, and system survivability, under various operational conditions and scenarios. However, due to the high computational cost, the simulation process of a complex system may become intractable and is often limited by the available computing power and time. One straightforward solution to this issue is to increase the computing power or resort to a distributed computing environment. However, this approach is often not an affordable solution and at times, does not offer sufficient reductions in execution time.

An alternative to improving the capability of the hardware associated with the simulation is to reduce the computational time through the use of surrogate models. Surrogate modeling techniques, which are particularly popular in the field of design and optimization, have been successfully applied to various analyses that require repeated and intensive computations and can save a significant amount of simulation time and cost in the design and optimization processes.

The sole purpose of surrogate modeling approach is to gain a huge improvement in computation speed of a model. As the name implies, a surrogate model is an approximation or a reduced-order model of a model. There are various types of surrogate models, which are from simple polynomial-based fit models to more complicated kernel-based or basis function-based models that are capable of higher-fidelity modeling, and the details of different surrogate modeling approaches are revisited in Chapter 2.

In that sense, a surrogate modeling method may be a good choice for the M&S of complex systems for the following reasons, which are that:

- Surrogate models can enable to expand the size of the design space exploration significantly.

- Design engineers can perform 'what-if' type analyses more rigorously and efficiently.

- Surrogate models are easily run in parallel or distributed computing environments.

- Surrogate models can be integrated to various M&S environments without technical challenges.

- Surrogate models are portable and license-free. The models can be translated into generic programming or scripting languages so that they can be distributed and run without expensive commercial tools.

There are also drawbacks in applying surrogate modeling methods, which are:

- Curse of dimensionality [13]. Surrogate modeling methods require the generation of a data set consisting of input-response pair samples through computer experiments. This sampling process becomes computationally impossible when the dimension of the design space is too large, since the number of computational cases for generating the data set grows combinatorially with respect to the growth of the input dimension.

- Surrogate models are incapable of containing the information of the system topological configuration, which is very important for the design of engineering networks.

- Some surrogate model is not suitable for approximating dynamic models or nonlinear models.

- The output of a surrogate model always contains error, compared to the output of the original model.

Based on the benefits and drawbacks above, a surrogate modeling approach can be considered as a great solution for the simulation cost, the third problem in §1.3. However, it can not address the first and second problems, because of its lack of modeling topological configurations of a system, and incapability to model the systems with a large number of inputs.

1.4.2 Bond Graph

Bond graph [84] is a graphical modeling method aimed for modeling and analyzing dynamic behaviors of multi-domain complex systems. This method is based on the idea that the behavior of most engineering systems – such as electric, mechanical, thermal, and hydraulic systems – can be commonly represented by power flows or energy exchanges between components inside those systems. In bond graph method, all power flows between components of a system are expressed by two variables called *flows* and *efforts*, but defending on the physical domain to which each component belongs, the specific attributes of flows and efforts are different. Table 2 introduces the flow and effort variables for different physical domains.

Figure 11 shows a simple example of a bond-graph model. The edge of the graph model is called *power bond*, or shortly, *bond*. A bond describes the power flow-based connection between two components in a system. The direction of the power flow through a bond is shown by a half-arrow placed at the either end of the bond.

There are two different types of junctions defined in bond graph reflecting two aspects of Kirchhoff's law. The '0' junction is the one representing Kirchhoff's current law so that the sum of all the flow variables is zero, but all the effort variables are the same in the junction. On the other hand, the '1' junction follows Kirchhoff's voltage law, meaning that all the flow variables are the same and the effort variables

Table 2: Effort and Flow Variables of Different Physical Domains in Bond Graph [1]

Systems	**Effort (e)**	**Flow (f)**
Mechanical	Force Torque	Velocity Angular velocity
Electrical	Voltage	Current
Hydraulic Thermal	Pressure Temperature Pressure	Volumetric flow rate Entropy change rate Volume change rate
Chemical	Chemical potential Enthalpy	Mole flow rate Mass flow rate
Magnetic	magneto-motive force	Magnetic flux

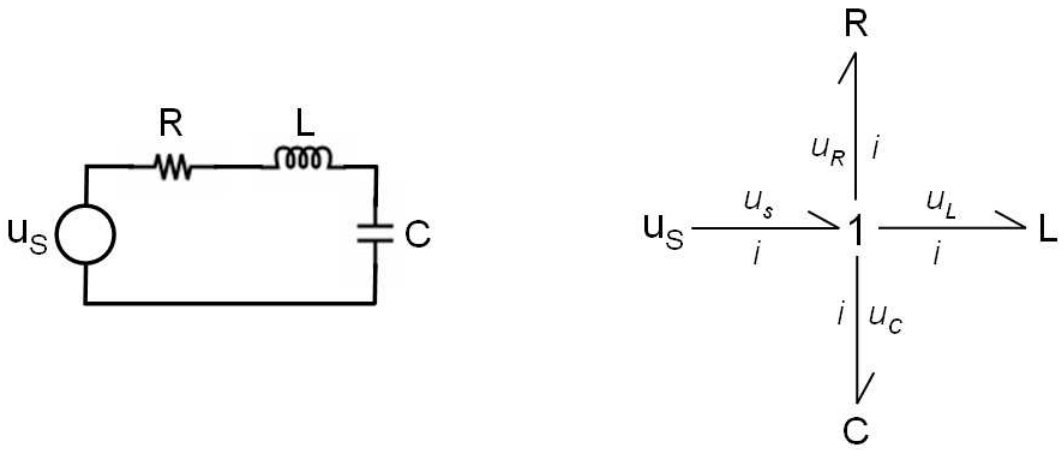


Figure 11: RLC Circuit and its Bond Graph (u: voltage, i: electric current)

are summed to zero. According to the notations of the two junctions, the bond graph in Figure 11 is made of a single junction of Kirchhoff's voltage law.

The energy-based representation of bond graph is especially beneficial for modeling multi-physics systems. The advantages of the bond graph approach are:

- A multi-physics, multi-domain system can be modeled by a common modeling formalism and method.
- Bond graph method is inherently capable of modeling dynamic systems.

- It is easy to apply an object-oriented modeling approach which enables higher reusability, adaptiveness, and maintainability of a model.
- Bond graph modeling naturally leads the users to systematic approaches for setting variables, interfaces, and processes.

However, there are also the disadvantages, which are:

- A bond graph model does not resemble the physical topology of an actual system.
- Bond graph's graphic formalism is difficult to understand and become too complicated to comprehend even for a relatively simple system (see Figures 12 and 13 for example).
- A bond graph model has relatively poor capability of representing the nonlinear components creating discontinuity, such as switches and diodes.
- A bond graph model is not suitable for automated model reconfiguration because the model is generated graphically (thus, manually).
- The bond graph approach does not necessarily provide the benefit of computational speed.
- There are not many bond graph-based modeling tools that are well maintained and updated in either commercial or open-sources sectors.

Regarding the three problems in §1.3, the bond graph method allows manual modifications of a model when damage modeling and design-oriented model reconfigurations are needed. For the computation efficiency, bond graph would not give any particular benefit.

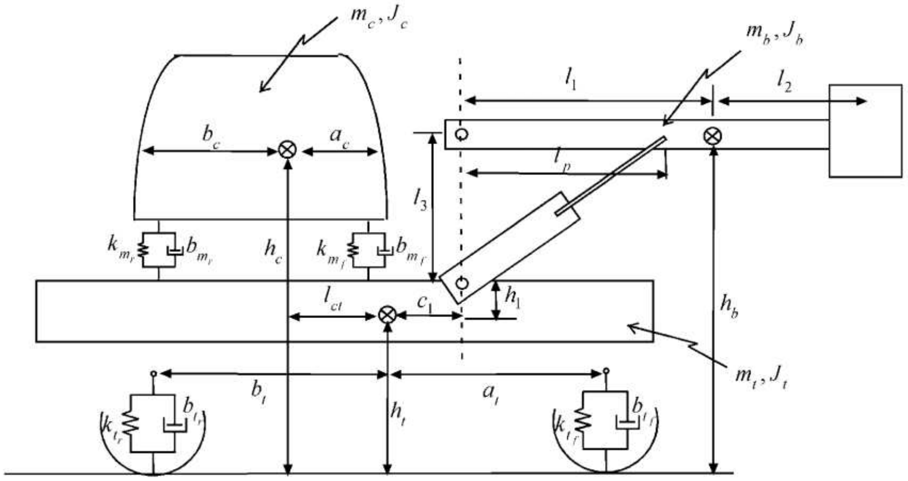


Figure 12: Backhoe Model (from Margolis and Shim [47])

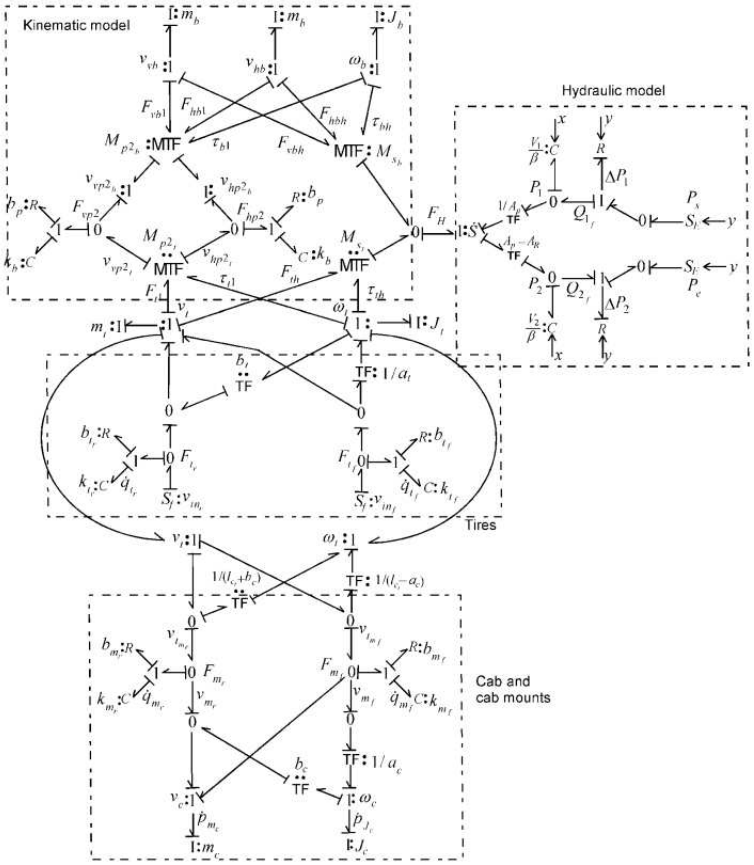


Figure 13: Bond Graph of Backhoe Model (from Margolis and Shim [47])

1.4.3 Component-Based Acausal Modeling

Efforts of building models of systems based on their components or modeling blocks are motivated by a simple goal – improving model reusability. Many different engineering systems share the same or similar components. Even for a single system, there are some parts that are used repeatedly in multiple places in the system. Therefore, an M&S environment with the libraries of highly reusable component models can improve modeling efficiency and productivity significantly.

There are many component-based modeling softwares, and they can be classified as either causal or acausal types, based on their ways of defining the component representation. In causal modeling tools, a component represents a functional process, which has the distinct definitions of input and output ports, just like a function in typical programming codes. An example of the component-based causal modeling tools is Simulink®, which may also be one of the most commonly used M&S software tools in many different scientific and engineering fields, and Figure 14 is an example diagram of Simulink models. Causal modeling tools are especially convenient for modeling control systems, some rigid-body dynamics that can be represented by ordinary differential equations, numerical computing algorithms, communication networks, and logistics systems, in which many physical or notional components also represent certain functional processes.

However, when it comes to modeling flow-based engineering network systems such as electric and fluid systems, the causal modeling approach can make the model generation very inefficient and difficult. The RLC circuit in Figure 11 is a good example. The components in the RLC circuit can be modeled by the following equations:

$$\begin{aligned} \text{Resistor:} &\qquad u_R = i_R R \\ \text{Inductor:} &\qquad L\frac{di_L}{dt} = u_L \\ \text{Capacitor:} &\qquad C\frac{du_C}{dt} = i_C \\ \text{Voltage source (input):} &\qquad u_S \end{aligned} \tag{1}$$

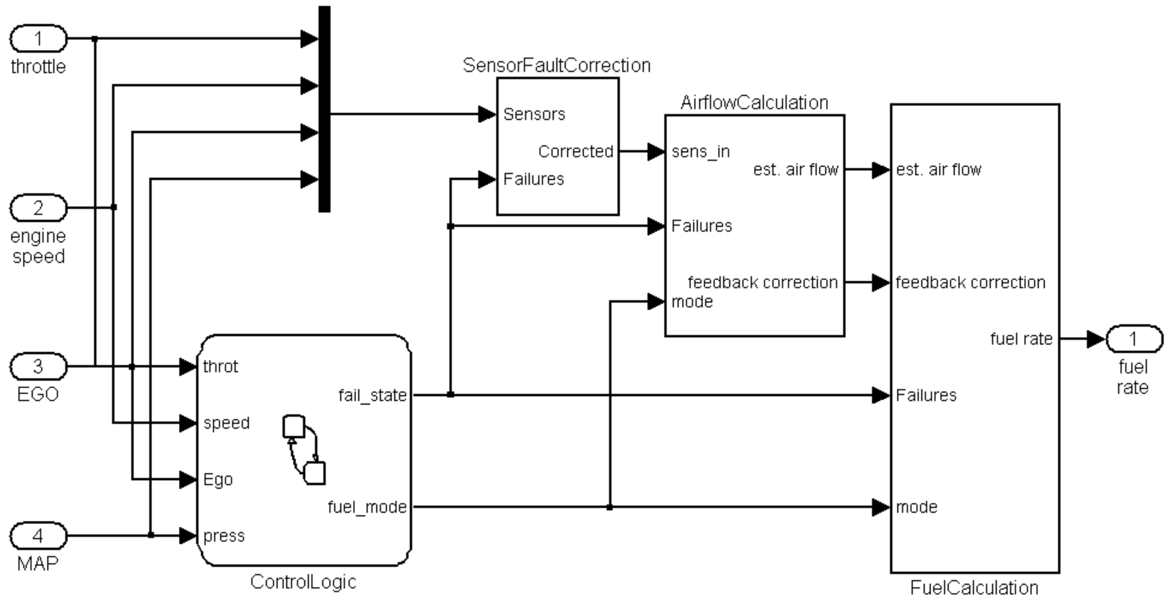


Figure 14: Simulink Model of Fuel Rate Controller of Automotive Engine (from Mathworks [48])

The connections provide the algebraic constraints of the system, which are:

$$\begin{aligned} &\text{KCL:} \qquad i_R = i_L = i_C \\ &\text{KVL:} \qquad u_S = u_R + u_L + u_C \end{aligned} \tag{2}$$

These components and their physical connections do not have any explicit input and output definitions. The set of system equations from Equations (1) and (2) is called a differential algebraic equation (DAE). In order to model the RLC circuit in Figure 11 with the causal modeling approach, the causality (input and output definitions) of each component should be chosen by a modeler, but the problem is that this causality may be changed when the connections among the components are changed.

One way is that the DAE of the RLC circuit is reorganized in the form of ODE, such as the one given in Equation (3), by a series of substitution of Equations (1) and (2), to create a monolithic model, but in this case, the model is only valid for a system with the topology in Figure 11, so has very limited reusability.

$$\begin{aligned}\frac{di_L}{dt} &= -\frac{R}{L}i_L - \frac{1}{L}u_C + \frac{1}{L}u_S \\ \frac{du_C}{dt} &= \frac{1}{C}i_L\end{aligned} \tag{3}$$

The modeling tools based on the acausal modeling approach address the problem of causality in modeling the flow-based systems. The core of a modeling tool based on the acausal modeling approach is a combination of a DAE solver and a sophisticated symbolic automation algorithm for finding a proper causality of a model. Thus, modeling engineers can consider only the system configuration when generating a model. This acausal modeling approach may be the current state-of-the art in component-based modeling and has been implemented in *Modelica*™[51], an object-oriented modeling language. Flowmaster® is also an acausal modeling tool in this perspective, although this tool is only for fluid system modeling.

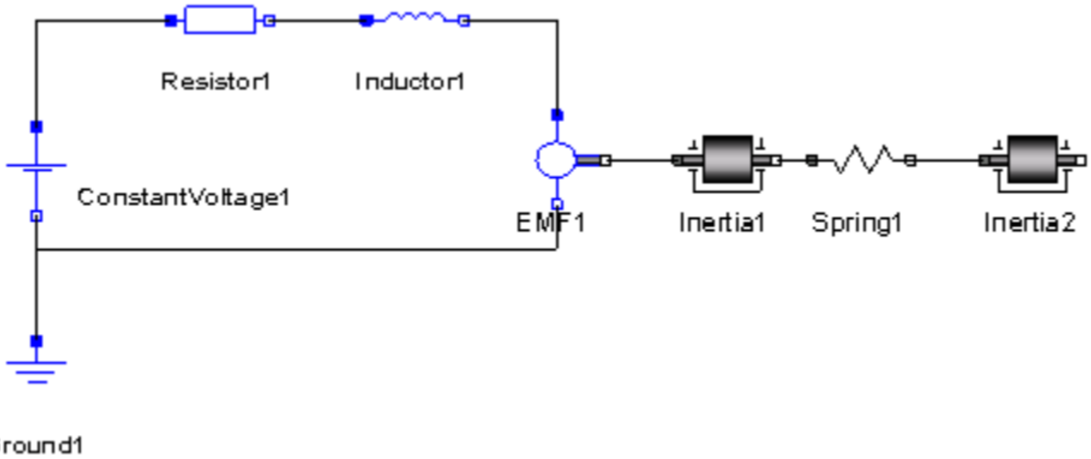


Figure 15: Modelica Model of DC-Motor with Spring and Inertia (from Fritzson [25])

Actually, *Modelica* is more than just a component-based acausal modeling tool. It pursues the unified modeling environment for modeling systems that are composed of different physical domains based on the *energy-based system modeling method*, which has, in fact, almost the same conceptual basis as the bond graph method. In Modelica, all the energy-based components have *flow* and *effort* variables as the universal variables across different physical domains.

Modelica is a complete object-oriented, script-based modeling environment, which gives great usability, flexibility, and efficiency of modeling. Modelica is also an open-source tool, which means there are a large number of model libraries that are available for free. Unlike the very rigid and obscure graphical formalism of bond graph, each basic component of Modelica is created by scripting. Thus, the creation of derivative component models is easy, and the connection of components is a lot like the physical connections inside an actual system, which makes building a complex system model become easier. Figure 15 is a simple example of a Modelica model, which has both the electric and the mechanical portions of the model, in the graphical model editing environment.

In outline, the advantages of modeling the ship fluid system using Modelica are as follows.

- Modelica provides both graphical and script-based environments for modeling.
- The graphical model highly resembles the connection topology of an actual system.
- The automation for component-level model reconfiguration is possible to be programmed.
- The acausal and energy-based modeling approaches can help the rapid creation of a complex enigneering network model significantly.
- Its object-oriented language can improve model reusability and expandability significantly.
- It is capable of dynamic simulations.
- It provides various built-in component libraries.

And the disadvantages of Modelica are:

- The component-based acausal modeling approach does not necessarily provide the benefit of computational speed.
- It does not support the changes of the model topological configuration during simulation. This is an important feature for modeling damage.
- A steep learning curve. Modelica language is still not popular in engineering and scientific fields, thus many modeling engineers may have to learn Modelica to start with modeling implementation.

Based on the three problems in §1.3, the component-based acausal modeling approach can partially address the problem of model reconfiguration, but it is still lack of the capabilities for solving the problems of damage modeling and computational cost.

1.4.4 Graph Theory

Graph theory, also called network theory or circuit theory in some disciplinary fields, has very diverse applications in many different disciplinary fields. In graph theory, a graph refers to a mathematical object formed with points and interconnections between them [30, 2]. There are a lot of systems that can be represented with nodes and connections in our world. Examples are physical networks such as fluid, electrical, road, and computer networks, or more abstract networks such as economic chains, human networks, and ecosystems. From the observations of broad applications, Evans and Minieka even found the great potential of graph theory, as an integration environment or "unifying basis from which results from other fields can be collected, shared, extended, and disseminated [23]."

As briefly mentioned, a graph is made of two elements, vertices (or nodes) and edges (or lines). In usual graph implementations, an edge represents a notional or physical route of the flow of (data) signal or physical medium between two nodes. On the other hand, a node represents either a functional process or just a junction point of incident edges, depending on its application domain. An example of a graph is shown in Figure 16.

In order to describe the behavioral states of a system, graph theory employs two different types of variables called the through variables and across variables. In general, through variables represent flow properties held on edges, and across variables represent potential properties on nodes. The notion of through and across variables seems similar to that of flow and effort variables in bond graph, but the actual attributes of through and across variables of a system are not necessarily the same as the attributes used for flow and effort variables of a bond graph representation of the same system [75].

The great strength of graph theory is in its various matrix tools for representing a graph in the numerical format. an example among these matrix representations is the incidence matrix, which describes the connectivity between vertices and edges in a graph. Equation (4) is the incidence matrix of the linear graph representation in

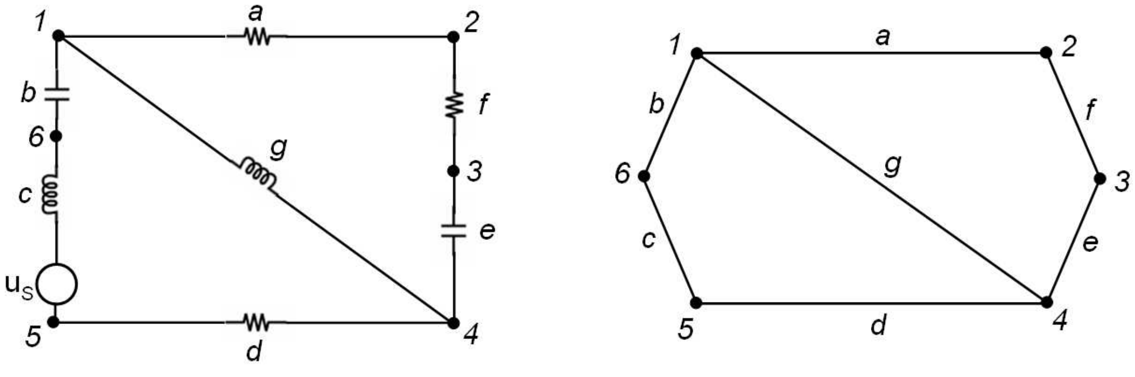


Figure 16: Simple Electric Network and Corresponding Linear Graph [16]

Figure 16.

$$A = \begin{array}{c|ccccccc|} & a & b & c & d & e & f & g \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 4 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 6 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array} \tag{4}$$

Letting i indicate the ith row, and j indicate the jth column of an incidence matrix, the ith row represents the ith node, and the jth column represents the jth edge of a graph. For the graph in Figure 16, the element of matrix A, $a_{ij} = 1$ if the ith node and the j edge are incident with each other, and $a_{ij} = 0$ if they are not. In addition to the incidence matrix, Graph theory provides a number of different matrices representing different aspects of the topological properties and characteristics of a graph. More details of graph theory are also given in Chapter 4.

The matrix representation of graph theory is a very useful and important feature for modeling physical networks, because a matrix form can be applied to computer programming very easily and efficiently. Using those matrix formats, the automation for generating computer models of different topologies can be made systematically and flexibly, but still not requiring a high level of coding skills.

Based on the observations above, the benefits of the Graph theory approach can be summarized as follows:

- A graphical representation of a model resembles the physical topology of an actual system.

- The matrix representations of a graph model can facilitate the automation of both damage modeling and model reconfiguration significantly.

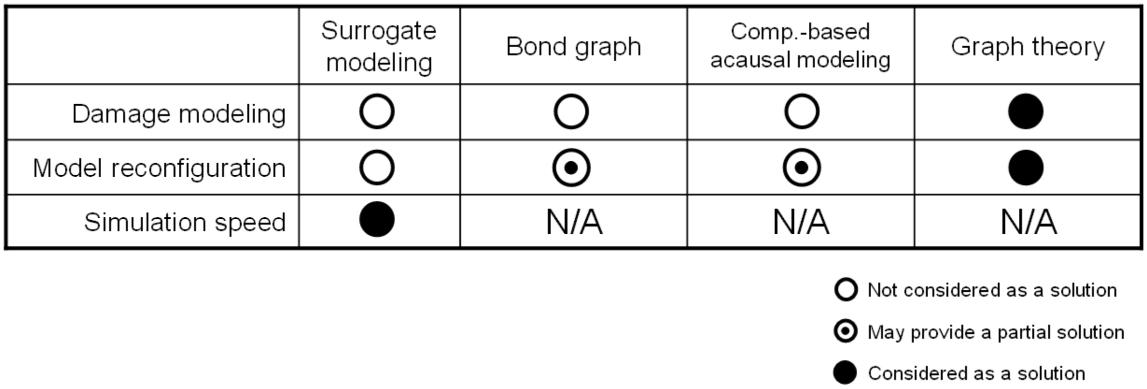
- Incorporated with the matrix representation, there are a large number of resources of analytic and numerical theories, methods, and techniques available for design and optimization of a graph model.
- Graph theory naturally leads the users to systematic approaches for setting variables, interfaces, processes, and model decomposition.
- It allows for multi-physics, multi-domain system modeling.

As for the shortcomings of the Graph theory approach, there seems just one issue, which is also significant, to point out. Graph theory is suitable for studying the properties and characteristics of the structure or topology of network-oriented problems. However, graph theory does not provide the modeling formalism or method for analyzing the physical behavior inside the network. In other words, graph theory alone does not provide the complete framework of modeling a certain physical system, so a behavioral modeling approach should be developed separately for modeling implementation.

Considering the three problems in §1.3, graph theory can provide great tools for addressing both the problems of damage modeling and model reconfiguration. Graph theory is not a complete solution, but can be useful ingredients of modeling an large-scale engineering system. This evaluation about graph theory is also consistent with Tam's conclusion in his review on various M&S approaches for large-scale reconfigurable engineering systems, stating that one of the bases for the M&S tool development for such systems should be linear graph representation [75].

1.4.5 Summary of Comparisons Based on Research Problems

Figure 17 shows the comparisons summary of the methods and approaches investigated in the previous sections, with respect to the three problems identified from the capability gaps that domain specific modeling tools or models have in an integrated M&S environment for a naval ship system.



	Surrogate modeling	Bond graph	Comp.-based acausal modeling	Graph theory
Damage modeling	○	○	○	●
Model reconfiguration	○	⊙	⊙	●
Simulation speed	●	N/A	N/A	N/A

○ Not considered as a solution
⊙ May provide a partial solution
● Considered as a solution

Figure 17: Comparison of Modeling Approaches/Methods Based on Three Problems

Figure 17 clarifies that no single approach in the investigated methods and approaches is capable of addressing all the three problems, but a combination of some of the methods and approaches, instead of a single one, may be a solution of the problems. Based on Figure 17, such a combination can be of surrogate modeling and graph theoretic approaches. Therefore, the two approaches are chosen and used as the conceptual basis for developing and formulating a solution approach of the three problems in this thesis.

1.5 Research Scope and Objective

Based on the problems identified previously in the integrated M&S of a naval system, the research aims at developing a modeling method that is more suitable for performing damage or failure analyses in a design process, especially in an early design phase. The scope of modeling is limited to individual domain models, particularly types based on physical flow-based networks, such as fluid or electric power networks. Thus, developing a method of integrating the domain specific models or signal-based network modeling is outside of the research scope. Considering the research goal, the successful development of the modeling method must solve the following challenges, which are served as the research goals:

1. Model reconfiguration should be highly flexible and automation-friendly for iterative design analyses.

2. Damage scenarios must be taken as the input to a domain M&S environment.

3. In order to handle a large number of simulation cases, models must be computationally affordable.

4. Because of limited human and temporal resources, modeling should avoid intensive involvement on deep expert knowledge of domain disciplines and coding skills.

The development of the M&S method is validated by the implementation of the M&S environment as the end-result. Although the application platform was chosen as a military ship, the method is generic and can be applied to any engineering platform that shares similar design paradigm – fail-proof design, design for reduced manning, maintenance, and operational cost by high-level of autonomy or automation of the system. In the thesis, the formulation is done for a fluid system which may be frequently found in many complex engineering products, but the resulting M&S formulation is expected to apply with minor modifications to electric power systems, another very popular type of systems found in complex engineering systems, due to their strong analogy to fluid systems.

1.6 Overview of M&S Formulation

Figure 18 describes the overview of the solution approach, which is constructed upon two key ideas – topological modeling based on the digraph representation and component behavior modeling with a neural net-based surrogate modeling technique. From these two key concepts, various methods, tools, and techniques are developed as the elements for formulating a M&S approach that addresses the research problems. The two concepts and the elements developed from them for the formulation of the solution approach are briefly explained in the following subsections.

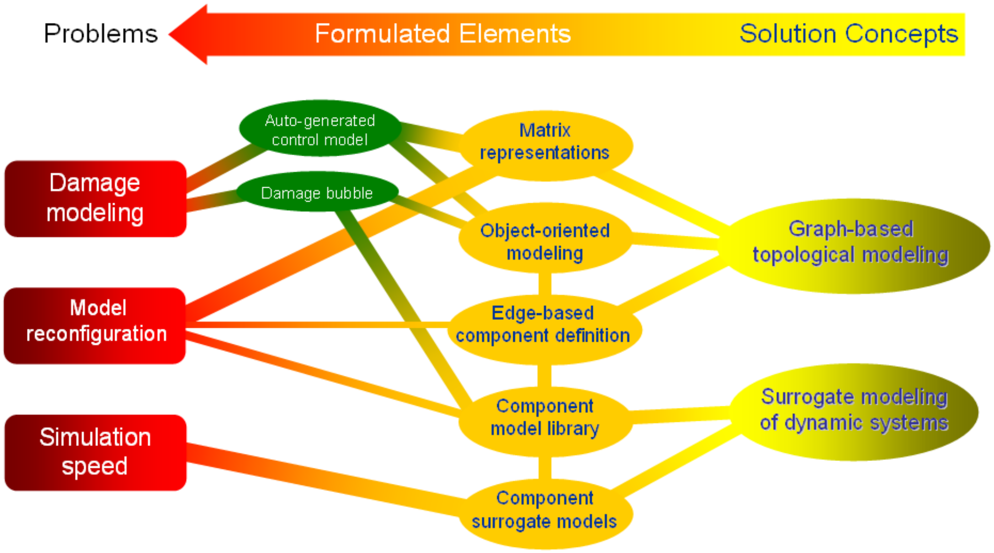


Figure 18: Formulated Solution Approach for Identified Problems of Domain M&S

1.6.1 Graph-Based Topological Modeling for Flexible Model Reconfiguration

Borrowed from the basic notion of Graph theory, a fluid network is represented by a graph. In the numerical implementation, the graph is a composite object made of edge and node objects and the incidence matrix as an attribute containing the information of the connectivity between nodes and edges. All subsystems or components of a fluid network are defined by corresponding edge objects in computer modeling, and this composite object is called the topological model of a fluid network by the author. In this modeling architecture, changes of the connection topology can be reflected simply by changes of the model's incidence matrix which should be automatically generated by extracting the local connectivity information in all edges that forms the current graph model. Reminding of numerical versatility of matrices, the graph-based topological modeling can be a great enabler for automated manipulation of the system topology for modeling damage and generating models of design alternatives.

As the name implies, a topological model describes only the connection topological structure of a fluid system without the behavior – which is often dynamic – of the system. In order to model the physical behaviors of the system, a number of behavioral models for different types of edge components of the fluid system model are created and used to model the physical behaviors inside edge components. Numerically, these behavioral models are implemented in the form of functions in Python, and a group of the model functions is managed as a component model library. The strategy of managing the topological model and the component models separately is especially beneficial for increasing the system model's scalability and plug-and-play capability, which are all crucial for generating models of different design alternatives in automation.

1.6.2 Surrogate Modeling of Dynamic System Components

The M&S approach included the additional, somewhat unusual setting, in which the behavior models in the component model library are created by a surrogate modeling technique for dynamic systems. This requires the presumed condition that there are existing component models from which surrogate models can be built.

Then, is this approach really appropriate? This can be answered by explaining why the other possible approaches are not feasible solutions. One straightforward approach is creating all component models with an available domain M&S tool. If the computational cost of the domain M&S tool is low, this will be a reasonable choice, but often, it is not. Another approach may be hard-coding all the component models from scratch, and if the behavior of each component can be modeled by simple physics equations, this approach would be a solution that is both feasible and computationally affordable. The problem, however, is that components of a fluid system are not easy to model. Even a single valve on a pipeline exhibits highly nonlinear dynamics whose relation with different valve opening ratio, flow speed, and geometric properties can only be modeled based on empirical formulas and data, which lead to the necessity of either in-depth expert knowledge, more man hours, or the use of a domain tool.

In summary, the surrogate modeling approach was formulated as follows: first, component models were built using a commercial domain M&S tool or legacy tools; then these models were translated into surrogate models and instantiated using a general programming environment. By doing so, the new M&S scheme is computationally less expensive than the M&S with a domain tool, but still keeps the modeling effort and cost less than modeling from scratch. In other words, the surrogate modeling approach is an enabler for speeding up the fluid model when it is too slow to perform design analyses.

1.6.3 Development of Damage Modeling Tools

Damage analysis often becomes the main activity for designing a more survivable and resilient system, but the elements introduced so far are not fully capable of modeling damages of a fluid system. Therefore, additional development is unavoidable for addressing damage modeling. Compared to the modeling method of a fluid system based on the graph-based topological modeling and the component-based surrogate modeling approaches, damage modeling is more likely an art than a method, since its approach is very application specific. The added formulation for damage analysis to the developed modeling method contains two elements, which are *damage bubble* and *reference damage control model*.

The damage bubble entity represents an explosion that causes actual damages onto a system and the spatial properties of the explosion. In the numerical implementation, its role is to identify the components that are affected by this explosion and change the properties of them in accordance to the imposed damage.

Another damage modeling entity is the reference damage control model. A damage analysis of a self-reconfigurable fluid system without any damage control effort is meaningless, since the analysis would yield only a trivial solution of the total system failure, but the control system model is not always available in the early design phase. The reference damage control model is developed to do the role of an initial control design in this case.

However, the essence of the damage modeling approach in this research is in the algorithm of the automated generation of the reference damage control model. In the formulated M&S environment, the reference damage control model is automatically generated for a given graph model, whenever the graph model is regenerated or modified from its original configuration, so the simulation-based design process can still be highly automated for damage analyses.

1.6.4 Numerical Implementation Environment

As the numerical environment, Python [78] was used for implementing the developed modeling method. Python is an object-oriented scripting language, which is easier to use and maintain than other lower level object-oriented programming languages such as C++ or Java. With adding proper extensions to Python such as IPython console and NumPy library, which are also used in the M&S environment, Python also features a Matlab-like interactive shell and matrix data structures, which are very convenient for scientific computing. All these features helped a faster and more efficient development of the M&S environment.

Separately from the M&S environment implemented in Python, the generation of component surrogate models for the component model library were performed with Matlab® Neural-Network Toolbox. The component surrogate modeling method is developed based on the recurrent neural network (RNN) as the mathematical structure of the component surrogate models, which is introduced in Chapter 3 in detail. For the RNN-based surrogate modeling implementation, Matlab® Neural-Network Toolbox provides an environment for powerful and rapid generation of RNN models with its feature-rich, built-in libraries.

CHAPTER II

SURROGATE MODELING APPROACHES: IN THE VIEW OF MODEL STRUCTURE

2.1 Introduction

Many of computer simulations in the design or optimization efforts are computationally expensive. It is quite common that a single simulation takes from several minutes to even a few days to run. However in the analysis for design and optimization, many computation tasks require a large number of experimental simulation runs – from a few dozen to hundreds of thousands – to identify the effects of certain design or input variables have on the system responses, and the computational burden of a simulation model is one of the major reasons to compromise either model fidelity or rigorousness in the design and optimization processes.

One popular alternative to using computationally-expensive computer models is to generate the surrogate models of those original models. Surrogate models are based on a very simple idea. For a computer model, the functional relationship between certain input variables and their responses can be approximated by some simple mathematical or logical expression which is significantly cheaper to compute. Since such an expression, or a surrogate model, is an approximation of the original model, it is inevitable that the accuracy is compromised in some level, but by the proper selections of the modeling ranges and surrogate modeling method, this loss of accuracy can be acceptable in a practical sense, compared to a huge benefit of computation speed.

In addition to the computational advantage, a surrogate model can deliver a few additional benefits. Since a surrogate model is realized in a simple mathematical expression (a polynomial is one of good examples), it can be implemented virtually in any computational or programming environment with a trivial effort of coding. This means a surrogate model can have far greater portability and interoperability than its original model which is often created by using a certain domain modeling tools or more complicated programming. Thus, in the case that the modeling environment of a model is lack of linking or external-interface capability but the physics and mathematical expressions of the model is too complex to duplicate to another modeling environment, the surrogate model may find another niche as a useful solution to such an embarrassing problem.

2.1.1 Previous Works in Surrogate Modeling Approach

The currently available surrogate modeling methods can be categorized by their types of model structures. In linear parametric modeling, the polynomial regression modeling approach, also known as the response surface methodology (RSM) [53, 7, 60], is probably the most commonly used method because it generally uses a simple quadratic (or cubic) polynomial as the model structure and the model can be fitted by a simple algorithm like the linear least-squares method.

RSM is, however, not suitable for any problem with strong nonlinearity, which requires nonlinear modeling approaches for better model accuracy. A popular example of nonlinear modeling approaches is DACE (Design and Analysis of Computer Experiments) method by Sacks et al. [62, 22]. DACE method is an interpolation modeling framework specialized for deterministic computer simulations based on Gaussian process modeling which is also known as Kriging in the field of geostatics. Another popular example for nonlinear approaches is artificial neural networks (ANN or NN). Both modeling approaches are discussed in more detail within this section.

There are many design applications of various surrogate modeling methods. A few examples of them are: Mack et al. [44] applied RSM for the design case study of a compact liquid-rocket radial turbine; Queipo et al. [60] applied and compared a RS model, a Kriging (or DACE) model, and a radial basis function (RBF) network model in the design optimization of a rocket's liquid-propellant injector; Simpson et al. [71] and Jeong et al. [35] performed a nozzle and a wing-section design respectively using the Kriging modeling approach; Manik et al. [46] created a model of pavement construction qualities, and Scharl and Mavris [65] created a parameterized forces and moments model of aircraft using NN.

What can be observed from the earlier works is that the surrogate modeling methods have been mainly focused on static models rather than dynamic models since the system-level design and optimization is mostly performed with just static models or analyses. Although examples can be found for the dynamic surrogate modeling approaches (see Merwe et al. [77] for example), they are yet rare and are in need of many improvements compared to static surrogate modeling approaches and applications.

2.1.2 Approach Based on System Identification

In this thesis, the solution for dynamic surrogate modeling is not formulated from just the approaches available in the surrogate modeling communities but by combining them with the methods of system identification, which has very sophisticated and well-built theories and methods dedicated to dynamic systems and their identification.

In outline, surrogate modeling is comprised of two main steps, which are firstly a computer experiment for generating the training data and then the model-fitting by which a surrogate model is created from the training data. In fact, one can easily recognize that the process of surrogate modeling is very similar to a usual process of statistical regression except that the data is not from a real system but the model of the system. In other words, a surrogate modeling method can be viewed as just problem-specific recollections or reorganization of more general studies such as regression analysis, approximation theories, or data analysis. The approach of implanting system identification to surrogate modeling keeps the same skeletal two-step process while providing elements that a static surrogate modeling is lacking, such as more delicate choices of model structure, better consideration of model stability, and the experimental design that is more proper for dynamic system modeling. And first of all, some available methods and theories of those aspects, particularly the model structure, is briefly reviewed in the rest of the chapter.

2.1.3 About System Identification

System identification is the subject of constructing or selecting mathematical models of a dynamic system based on measured data [40, p.1][41, p.79].

According to the historical summary of system identification by Gevers [26], system identification was originated from the field of statistical time-series analysis, which became the reason that the literatures of system identification share many jargons of statistical time-series analysis, such as AR, MA, ARX, and ARMAX, to name a few. In 1960s, with the prosperity of modern control theory and the blooming new theory of model-based control design, the scientific communities began to fuse control theories with data-based system estimation approach, which led to the birth of system identification as a distinct engineering field. Since then, system identification has been a very important tool set for control engineering, as well as scientific simulation, modeling, prediction, fault detection, etc. [8].

In this thesis, models are assumed to be black-box models, which means that a modeling engineer has no a priori knowledge of the mathematical structures and physics of the models and their processes. There is also a gray-box model of which some of physics and mathematical structure is known so the identification problem becomes to find only a few unknown parameters. A white model refers to the models that is perfectly known in its physics and model structure. Most theories and methods of system identification assumes a discrete-time model instead of a continuous-time model since a system produces the discrete data in practice. Therefore, only a discrete-time system model is considered in this paper.

The following overview of the theoretic basics of system identification is largely based on Ljung [41], Ljung and Glad [42], Janczak [34], and Pearson [59]. Additional references for more specific topics are: Schetzen [67] and Ogunfunmi [57] for §2.3.1 and 2.3.4, and Sjöberg [72] for §2.3.3.

2.2 Overview of Linear Model Structures

For a linear system, a relation between the discrete inputs and outputs may simply be expressed by the following linear difference equation:

$$\begin{aligned} y(t) + a_1 y(t-1) + a_2 y(t-2) + \ldots + a_{n_a} y(t-n_a) = \\ b_1 u(t-1) + b_2 u(t-2) + \ldots + b_{n_b} u(t-n_b) + e(t) \end{aligned} \tag{5}$$

where $u(t)$ and $y(t)$ is the system's discrete input and output at time t and $e(t)$ a white-noise term representing model or equation error.

Equation (5) can be reorganized in the following way when it is used as a predictor of a system output based on the previous data.

$$\begin{aligned} y(t) = -a_1 y(t-1) - a_2 y(t-2) - \ldots - a_{n_a} y(t-n_a) \\ + b_1 u(t-1) + b_2 u(t-2) + \ldots + b_{n_b} u(t-n_b) + e(t) \end{aligned} \tag{6}$$

Now, let us define θ and $\varphi(t)$ such that,

$$\begin{aligned} \theta &= [-a_1 \ \ldots \ -a_{n_a} \ b_1 \ \ldots \ b_{n_b}]^T \\ \varphi(t) &= [y(t-1) \ \ldots \ y(t-n_a) \ u(t-1) \ \ldots \ u(t-n_b)]^T \end{aligned}$$

Then, Equation (6) is now expressed as,

$$y(t) = \theta^T \varphi(t) + e(t) \tag{7}$$

In Equation (7), θ is called the parameter vector, and $\varphi(t)$ the regression vector of a linear black-box model. The elements of vector $\varphi(t)$ are called regressors. For the linear model in Equation (7), the model identification problem becomes to find the estimates of the unknown parameters θ with which the model best approximates the system within the given data. As an algorithm for solving this problem, the most straight forward example is the linear least squares method. Since the identified model is just an approximation of the system, a linear model with a known estimate of the vector θ is,

$$\hat{y}(t|\theta) = \theta^T \varphi(t) \tag{8}$$

For a linear model, a transfer-function form is especially preferred in the control communities because this form is especially convenient for many other applications in controls, such as response characteristics analysis, stability analysis, and control design. In order to obtain a simpler example, let us assume the error term in Equation (6) negligible, then the difference equation can be rewritten in the transfer-function representation,

$$\begin{aligned} y(t) &= G(q)u(t) \\ &= \frac{b_1 q^{-1} + b_1 q^{-2} + \ldots + b_{n_b} q^{n_b}}{1 + a_1 q^{-1} + a_2 q^{-1} + \ldots + a_{n_a} q^{n_a}} u(t) \end{aligned}$$

where q^{-1} denotes the backward time shift operator, which is $y(t-1) = q^{-1}y(t)$. There are various linear model structures, which can be efficiently described using the transfer-function representation.

2.2.1 Auto-Regressive with Exogenous Input (ARX) Model

The linear model in Equation (6) is called an *ARX* model, which is one of the most commonly used model type in the linear system identification. The regressors of the ARX model are from the tapped delayed signals from the actual system output and input as shown in Figure 19(a), and because of its structure, the ARX model is often recognized and used as a prediction model. The transfer-function representation of the ARX model in Equation (6) is slightly different than the above example because of the error term so that,

$$A(q)y(t) = B(q)u(t) + e(t) \tag{9}$$

or,

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{1}{A(q)}e(t) \tag{10}$$

The error term $e(t)$ in the equation is not important if the stochastic effect in $e(t)$ is insignificant (i.e., the system is dominantly deterministic) so the term with the error is often converted to the expression with the actual system output in practice, by inserting $e(t) = y(t) - \theta^T\varphi(t) = y(t) - \hat{y}(t|\theta)$ in Equation (9). Then, the transfer-function of the ARX model can be expressed as,

$$\hat{y}(t|\theta) = B(q,\theta)u(t) + (1 - A(q,\theta))y(t) \tag{11}$$

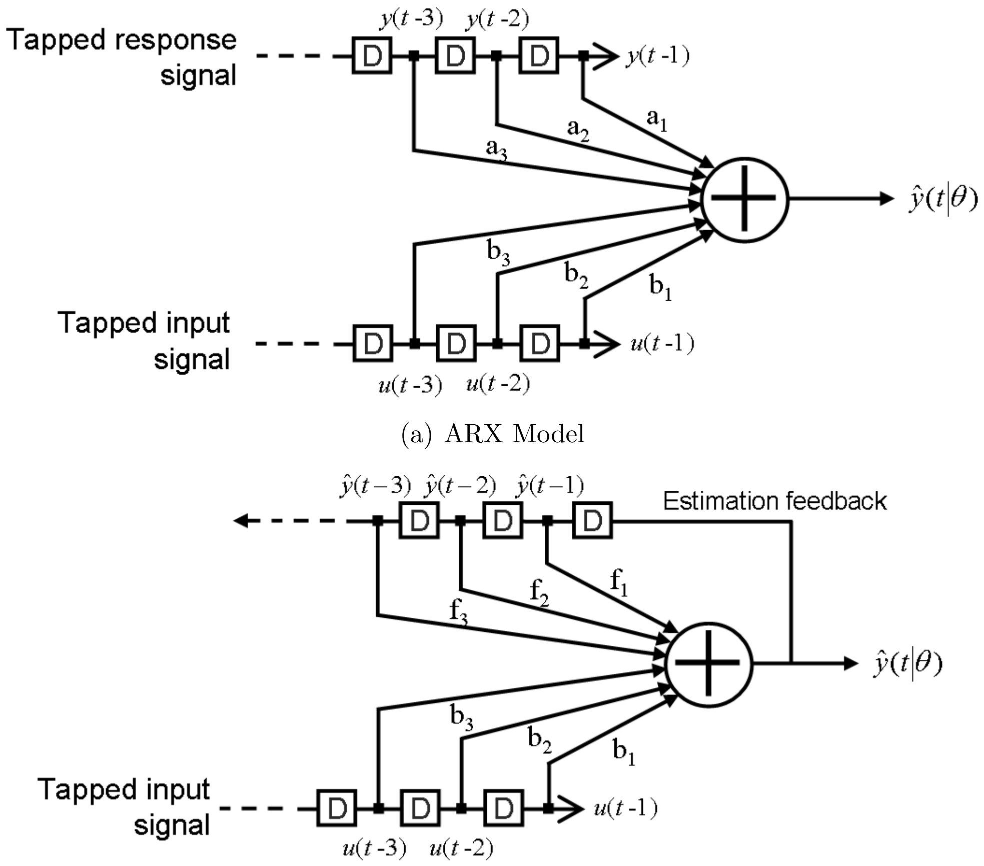
which is coincident with the linear difference equation in Equation (6).

2.2.2 Output-Error (OE) Model

The linear difference equation of OE model is given in Equations (12) and (13). Figure 19(b) also shows the structure of OE model.

$$\begin{aligned} &\hat{y}(t) + f_1\hat{y}(t-1) + f_2\hat{y}(t-2) + \ldots + f_{n_f}\hat{y}(t-n_f) \\ &\qquad = b_1u(t-1) + b_2u(t-2) + \ldots + b_{n_b}u(t-n_b) \qquad (12) \\ &y(t) = \hat{y}(t) + e(t) \qquad (13) \end{aligned}$$

As described in Equation (12) and Figure 19(b), the structural difference of the *OE* model from the ARX model is that its regressors are the delayed signal of the actual input as well as the feedbacks of its own output estimates, instead of the real output data from the system for the ARX model. Because of its structure, the OE model is often recognized and used as a simulation model.



(a) ARX Model

(b) OE Model

Figure 19: Linear ARX and OE Models

In the problem of identifying the OE model, the parameter and the regression vectors are defined by,

$$\theta = \left[-f_1\ \dots\ -f_{n_f}\ b_1\ \dots\ b_{n_b}\right]^T \tag{14}$$

$$\varphi(t) = [\hat{y}(t-1|\theta)\ \dots\ \hat{y}(t-n_a|\theta)\ u(t-1)\ \dots\ u(t-n_b)]^T \tag{15}$$

Based on Equations (12) and (13), the transfer function representation of the OE model is,

$$y(t) = \frac{B(q,\theta)}{F(q,\theta)}u(t) + e(t) \tag{16}$$

or the OE model can also be described by

$$\hat{y}(t|\theta) = \frac{B(q,\theta)}{F(q,\theta)}u(t) \tag{17}$$

The OE model structure that has the feedback of the outputs from itself as its regressors make the identification problem of the OE model more complicated than the ARX model because, unlike the ARX model which presumably has the true output $y(t-1), \ldots, y(t-n_a)$ as the given data, an OE model is only accessible to output estimates $\hat{y}(t-1|\theta), \ldots, \hat{y}(y-n_f|\theta)$, and the true output values are unknown. As a result, the identification problem has to be solved by a recursive learning algorithm which takes significantly more computation time than batch learning algorithms. Since the error term $e(t)$ is also unknown, the choice of the initial value of $\hat{y}(t-1|\theta), \ldots, \hat{y}(y-n_f|\theta)$ becomes another nuisance problem in both the identification problem and its application, if the effect or magnitude of $e(t)$ is significant. However for the deterministic problem where the effect of $e(t)$ is negligible, the structure of the OE model is more naturally suitable as a simulation model because most simulation models contain the direct feedbacks of their own outputs in order to model the dynamics of their systems, as the OE model does.

2.2.3 Other Models

Although not considered as important elements in the contexts of the thesis, several other linear model structures – particularly FIR, ARMAX, and BJ models – are briefly noted as a background information that may help readers understanding the reasoning behind the flow of the thesis.

2.2.3.1 Finite Impulse Response (FIR) Model

The FIR model has the simplest structure among all the models presented here. The difference equation of the FIR model is described as

$$y(t) = b_1u(t-1) + b_2u(t-2) + \ldots + b_{n_b}u(t-n_b) + e(t) \tag{18}$$

FIR models have only the time-series of the discrete input signals as the regressors so no feedbacks are needed for both identification and simulation/prediction uses.

Inherently, this model structure is BIBO stable because the response is a linear combination of the finite number of input samples. The FIR model performs well for modeling a system with very fast impulse responses for which a reasonably small number of input samples are enough; however, it is not a proper choice if the system has a slow dynamics (i.e. a very small time constant) because it would need too many delayed input samples as the regressors, for a good estimation ability. And obviously, the FIR model is impossible to model unstable systems.

2.2.3.2 Auto-Regressive Moving-Average with Exogenous Input (ARMAX) Model

The ARMAX model is a general version of the ARX model with the added MA (moving-average) part for describing the variation of the output responses corresponding to history of the white noise-type error input. The difference equation of the ARMAX model is,

$$\begin{aligned} y(t) = &- a_1 y(t-1) - a_2 y(t-2) - \ldots - a_{n_a} y(t-n_a) \\ &+ b_1 u(t-1) + b_2 u(t-2) + \ldots + b_{n_b} u(t-n_b) \\ &+ c_1 e(t-1) + c_2 e(t-2) + \ldots + c_{n_c} e(t-n_c) \end{aligned} \tag{19}$$

And, in the transfer-function form,

$$y(t) = \frac{B(q)}{A(q)} u(t) + \frac{C(q)}{A(q)} e(t) \tag{20}$$

2.2.3.3 Box-Jenkins (BJ) Model

The BJ model is a generalization of the OE model, and it can be described by,

$$\begin{aligned} \mu(t) = &-f_1 \mu(t-1) - f_2 \mu(t-2) - \ldots - f_{n_f} \mu(t-n_f) \\ &+ b_1 u(t-1) + b_2 u(t-2) + \ldots + b_{n_b} u(t-n_b) \end{aligned} \tag{21}$$

$$\begin{aligned} w(t) = &-c_1 w(t-1) - c_2 w(t-2) - \ldots - c_{n_c} w(t-n_c) \\ &+ d_1 e(t-1) + d_2 e(t-2) + \ldots + e_{n_e} e(t-n_b) \end{aligned} \tag{22}$$

$$y(t) = \mu(t) + w(t) \tag{23}$$

The transfer-function representation shows the model structure more clearly.

$$y(t) = \frac{B(q)}{F(q)}u(t) + \frac{C(q)}{D(q)}e(t) \tag{24}$$

Also known as the auto-regressive integrated moving average (ARIMA) model in the statistical time-series communities, the BJ model is basically the combination of a OE model as the estimation of changes of the long-term trend of the output data and an auto-regressive moving-average (ARMA) model that describes the short-term stationary disturbance.

For a system without significant stochastic processes, a general model like ARMAX or BJ model just over-complicate the modeling problem. Since the application in this thesis is for surrogate modeling a deterministic computer model, such general models are less useful than the simpler ARX and OE models.

2.3 Overview of Nonlinear Model Structures

In the real world, the portion of linear systems is extremely smaller than that of nonlinear systems, or strictly speaking, there is no such a thing as a linear system but just a system with linear approximation. Despite of this fact, the studies of linear systems have been incomparably more popular than those of nonlinear systems because nonlinear systems are too hard to understand.

Since there is already the well-developed linear system theory, scientists and engineers naturally have tried to project some of the knowledges, notations, or concepts of the linear system theory to the study of nonlinear systems, and so has the field of system identification.

Many literatures in nonlinear system identification use the acronyms like *NARX* (‘*N*’ denotes ‘nonlinear’), *NOE*, *NARMAX*, and *NFIR*, borrowing from those of the linear system theory, as a notional classification of different nonlinear black-box models. However the way of classifying linear models seems not very effective for the nonlinear models since it only specifies the definition of the regressors which is just one aspect that characterizes nonlinear system models. What is more important is on the mathematical formulations and structures that tie the model parameters or coefficients and the regressors since a mathematical formulation of a model is the main factor that determines what kind of the qualitative behaviors the model can approximate. In this section, several typical model structures that are particularly popular in nonlinear black-box modeling are introduced with the discussions of their underlying mathematical formulations, strengths, and drawbacks.

2.3.1 Volterra Model

The Volterra series expansion as was first established by the mathematician Vito Volterra but was first applied for nonlinear-system modeling in 1942 by Norbert Wiener [67, p.7], who was also known as a founding father of the field of cybernetics. Since than, the Volterra series expansion has been one of the most popular nonlinear models [57, p.13].

For a time-invariant, SISO, continuous dynamic system, the mathematical mapping between inputs and outputs can be expressed using the continuous-time Volterra series that is

$$\begin{aligned} y(t) = h_0 &+ \int_{-\infty}^{\infty} h_1(\tau_1)x(t-\tau_1)d\tau_1 + \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_2(\tau_1,\tau_2)x(t-\tau_1)x(t-\tau_2)d\tau_1 d\tau_2 \\ &+ \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h_3(\tau_1,\tau_2,\tau_3)x(t-\tau_1)x(t-\tau_2)x(t-\tau_3)d\tau_1 d\tau_2 d\tau_3 \\ &+ \ldots \\ &+ \int_{-\infty}^{\infty}\ldots\int_{-\infty}^{\infty} h_n(\tau_1,\ldots,\tau_n)x(t-\tau_1)\ldots x(t-\tau_n)d\tau_1\ldots d\tau_n + \ldots \end{aligned} \tag{25}$$

where $n = 1, 2, 3, \ldots, \infty$, and the terms

$$h_n(\tau_1, \ldots, \tau_n) \tag{26}$$

are known as Volterra kernels, which becomes 0 if $\tau_i < 0$ for $i = 1, \ldots, n$. The functional form of the Volterra series looks similar to Taylor series, which implies that the Volterra series is theoretically capable of modeling any continuous dynamic systems, and it is frequently used as the model structuring basis for polynomial-based modeling.

In the experimental and digital computing environments, a discrete-time, truncated Volterra series is more useful than the analytic, continuous-time representation in Equation (25). The discrete, finite-order version is expressed by

$$\begin{aligned} y(t) = h_0 &+ \sum_{i_1=0}^{M} h_1(i_1)x(t-i_1) \\ &+ \sum_{i_1=0}^{M}\sum_{i_2=0}^{M} h_2(i_1,i_2)x(t-i_1)x(t-i_2) \\ &+ \sum_{i_1=0}^{M}\sum_{i_2=0}^{M}\sum_{i_3=0}^{M} h_3(i_1,i_2,i_3)x(t-i_1)x(t-i_2)x(t-i_3) \\ &+ \ldots \\ &+ \sum_{i_1=0}^{M}\ldots\sum_{i_n=0}^{M} h_n(i_1,\ldots,i_n)x(t-i_1)\ldots x(t-i_n) \end{aligned} \tag{27}$$

where $n = 1, 2, ..., N$. In Equation (27), N is called the nonlinear order, which indicates the number of terms of the Volterra series expansion, and M the dynamic order, which indicates the number of delays of the inputs to the Volterra model. For instance, assuming a SISO NFIR model, a Volterra model of $M = 1$ and $N = 2$ is,

$$\begin{aligned} y(t) &= h_0 + h_1(0)u(t) + h_1(1)u(t-1) \\ &\quad + h_2(0,0)u(t)^2 + h_2(0,1)u(t)u(t-1) + h_2(1,1)u(t-1)^2 \end{aligned}$$

and the problem becomes measuring six coefficients of the model.

The Volterra series expansion is a very powerful modeling tool that can model virtually any continuous models with reliable accuracy. However, the identification of the Volterra kernel is a difficult problem because the outputs from the terms of Volterra series are not separable since they do not generate orthogonal outputs. Another problem is that the model tends to diverge once the input value goes out of a certain range of the input. There is a limitation of the Volterra model in a practical point of view too. From Equation (27), it is easily recognizable that the Volterra model can go numerically too complex and expensive to estimate or use, even with a few dimensions of input variables or the several orders of M and N of the model, so that is not a feasible model in the case of MIMO/MISO, high-order discrete systems, which are often met in practice. Table 3 summarizes the pros and cons of the Volterra model as the model structure of surrogate modeling for dynamic systems.

Table 3: Pros and Cons of Volterra Model

Pros	Cons
• Theoretically capable of modeling any continuous dynamic systems.	• Identification of Volterra kernel is difficult. • Model diverges when inputs go out of a certain range. • Even with a few inputs or orders of M and N, the model becomes numerically too complicated and expensive to estimate.

2.3.2 Kriging Method (or DACE Method)

Kriging is a statistical prediction modeling method developed in the field of spatial statistics and geostatistics [29, 22]. Kriging method is also called Gaussian process modeling, interchangably, especially in the field of machine learning and data analysis [61].

In fact, what is called Kriging modeling method in the literatures of surrogate modeling is mostly the DACE (Design and Analysis of Computer Experiments) framework, which is an interpolation-based regression method proposed by Sacks at al. for generating approximated models from deterministic, static computer models, inspired from Kriging method [62]. Although, its original formulation is based on the assumption of a static system, the brief introduction of the method of Sacks et al. may be a good starting point to understand the underlying theoretic approaches of the Kriging-based surrogate modeling.

The Kriging model is expressed by,

$$y(x) = f(x)^T \beta + Z(x) \tag{28}$$

with

$$f(x) = [\ f_1(x)\ f_2(x)\ \ldots\ f_m(x)\ ]^T$$
$$\beta = [\ \beta_1\ \beta_2\ \ldots\ \beta_m\ ]^T$$

where the vector input $x \in \mathbb{R}^p$, $f(x)$ a vector of linear regression equations which can be, for instance, polynomials to just a constant mean value, and β a vector of the coefficients for the regression equations. Z(x) is the Gaussian random process from $N(0, \sigma^2)$ with its covariance calculated by

$$Cov[Z(x), Z(w)] = \sigma^2 R(\theta, x, w) \tag{29}$$

where $R(\theta, x, w)$ is the correlation between the two points x and w. At this point, σ and θ are unknown, and the correlation R is assumed to be a parameterized function of one-dimensional distance value of the two points.

This way of defining the correlation does a critical role of the formulation of Kriging method. As a prediction point x gets far from a stationary, known sample point w, the correlation of the known value at w for predicting at x becomes weaker, eventually going to zero. In opposition, this correlation will be stronger as the points gets closer. There are various choices of functions for estimating the correlation (see [62, 38] for various correlation functions), but the most frequently used one may be the exponential function expressed as,

$$\begin{aligned} R(\theta, x, w) &= \prod_{k=1}^{p} \exp\left(-\theta_k \left|x_k - w_k\right|^2\right) \\ &= \exp\left(\sum_{k=1}^{p} -\theta_k \left|x_k - w_k\right|^2\right) \end{aligned} \tag{30}$$

where θ_k, x_k, and w_k are the k-th elements of vectors θ, x, and w. There is also a simplified version which uses a scalar value of θ for all directions of the distance, and it is expressed as,

$$R(\theta, x, w) = \exp\left(-\theta \left\|x - w\right\|_{l_2}^2\right) \tag{31}$$

where, θ is a scalar, and the distance is just a Euclidean distance. Here, θ is the scaling parameter that adjust the gradient of the correlation decay. The correlation decreases more suddenly if θ gets larger, implying the two points have a weaker correlation.

The estimation of the model in Equation (28) is based on the response data from a system. From n sample points $s = [\ s_1\ s_2\ \ldots\ s_n\ ]^T$, the following data are created,

$$\boldsymbol{Y} = \begin{pmatrix} y(s_1) \\ y(s_2) \\ \vdots \\ y(s_n) \end{pmatrix}, \quad \boldsymbol{F} = \begin{pmatrix} f^T(s_1) \\ f^T(s_2) \\ \vdots \\ f^T(s_n) \end{pmatrix}, \quad \boldsymbol{Z} = \begin{pmatrix} Z(s_1) \\ Z(s_2) \\ \vdots \\ Z(s_n) \end{pmatrix} \tag{32}$$

where $\boldsymbol{F} \in \mathbb{R}^{n \times m}$ and $\boldsymbol{Y}, \boldsymbol{Z} \in \mathbb{R}^{n \times 1}$.

The correlation matrix $\boldsymbol{R}$ for all the sample points in s is defined by

$$\boldsymbol{R} = \begin{bmatrix} R(\theta, s_1, s_1) & R(\theta, s_1, s_2) & \cdots & R(\theta, s_1, s_n) \\ R(\theta, s_2, s_1) & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ R(\theta, s_n, s_1) & \cdots & \cdots & R(\theta, s_n, s_n) \end{bmatrix} \tag{33}$$

where $R(\theta, s_i, s_j)$ is the correlation function of any two sample points in s, which are calculated by the way that was described in (30). The correlation matrix $\boldsymbol{R}$ is symmetric and in $\mathbb{R}^{n \times n}$. Applying the covariance in Equation (29) and the correlation matrix $\boldsymbol{R}$, the covariance matrix of the whole data is

$$cov(\boldsymbol{Z}\boldsymbol{Z}^T) = \sigma^2 \boldsymbol{R} \tag{34}$$

Similarly, the correlation vector $r(x)$ for computing the correlation between the prediction point x and the data s is defined by,

$$r(x) = [\ R(\theta, x, s_1)\ R(\theta, x, s_2)\ \dots\ R(\theta, x, s_n)\ ]^T \tag{35}$$

and the covariance of the same points are,

$$cov(Z(x), \boldsymbol{Z}) = \sigma^2 r(x) \tag{36}$$

Now, the best linear unbiased predictor (BLUP) is obtained by formulating the linear predictor in a following way:

$$\hat{y}(x) = c^T(x)\boldsymbol{Y} \tag{37}$$

The BLUP is obtained by finding the function vector $c(x)$ that minimizes the mean squared error of $\hat{y}(x)$, i.e. by the least squares estimation for solving $c(x)$.

$$MSE(\hat{y}(x)) = E[\hat{y}(x) - y(x)] \tag{38}$$

Applying the assumption of the unbiased condition ($E[\hat{y}(x)] = E[y(x)]$), and with somewhat lengthy derivations – which is not covered here (see Lee and Jung [38, pp.8-12] for more details of deriving from (38) to (44)), the MSE can be expressed as

$$MSE(\hat{y}(x)) = Var[y(x)] + Var[\hat{y}(x)] - 2\ Cov[\hat{y}(x), y(x)] \tag{39}$$

Each variance and covariance terms are obtained by using Equations from (32) to (37) so the final expression of the $MSE(\hat{y}(x))$ can be,

$$MSE(\hat{y}(x)) = \sigma^2 \left(1 + c^T(x)\boldsymbol{R}c(x) - 2c^T(x)\boldsymbol{r}(x)\right) \tag{40}$$

and the solution that minimizes the $MSE(\hat{y}(x))$ is

$$c(x) = \boldsymbol{R}^{-1}\left[r(x) - \boldsymbol{F}(\boldsymbol{F}^T\boldsymbol{R}^{-1}\boldsymbol{F})^{-1}(\boldsymbol{F}^T\boldsymbol{R}^{-1}r(x) - f(x))\right] \tag{41}$$

By substituting $c(x)$ in Equation (37), the BLUP becomes

$$\hat{y}(x) = \left[r^T(x) - \left(r^T(x)\boldsymbol{R}^{-1}\boldsymbol{F} - f^T\right)\left(\boldsymbol{F}^T\boldsymbol{R}^{-1}\boldsymbol{F}\right)^{-1}\boldsymbol{F}^T\right]\boldsymbol{R}^{-1}\boldsymbol{Y} \tag{42}$$

Then by introducing

$$\hat{\beta} = \left(\boldsymbol{F}^T\boldsymbol{R}^{-1}\boldsymbol{F}\right)^{-1}\boldsymbol{F}^T\boldsymbol{R}^{-1}\boldsymbol{Y} \tag{43}$$

which gives a more clear expression of the estimator that is similar to Equation (28)

$$\hat{y}(x) = f^T(x)\hat{\beta} + r^T(x)\boldsymbol{R}^{-1}\left(\boldsymbol{Y} - \boldsymbol{F}\hat{\beta}\right) \tag{44}$$

In Equation (44), the first term represents a long-term, global approximation model, and the second term is for any short-term Gaussian disturbances. The prediction model is not yet completed because σ^2 is unknown and the matrix $\boldsymbol{R}$, the function vector $r(x)$, and the regression coefficient $\hat{\beta}$ are parameterized with θ which is unknown.

In order to find the values of the parameter set θ, the maximum likelihood estimation (MLE) performed. Since each response from the data is assumed to follow the Gaussian random process, the likelihood function is given as,

$$L = \frac{1}{(2\pi)^{n/2}\sqrt{\sigma^n\left|\boldsymbol{R}\right|}} e^{\frac{(\boldsymbol{Y}-\boldsymbol{F}\hat{\beta})^T\boldsymbol{R}^{-1}(\boldsymbol{Y}-\boldsymbol{F}\hat{\beta})}{2\sigma^2}} \tag{45}$$

By changing it to the log-likelihood function, Equation (45) can be expressed as,

$$\ln L = -\frac{n}{2}\ln 2\pi - \frac{n}{2}\ln\sigma^2 - \frac{1}{2}\ln\left|\boldsymbol{R}\right| - \frac{(\boldsymbol{Y}-\boldsymbol{F}\hat{\beta})^T\boldsymbol{R}^{-1}(\boldsymbol{Y}-\boldsymbol{F}\hat{\beta})}{2\sigma^2} \tag{46}$$

The estimate of σ^2 that maximizes $\ln L$ is obtained as,

$$\underset{\sigma^2}{\operatorname{argmax}}\left(\ln L\right) = \hat{\sigma^2} = \frac{(\boldsymbol{Y}-\boldsymbol{F}\hat{\beta})^T\boldsymbol{R}^{-1}(\boldsymbol{Y}-\boldsymbol{F}\hat{\beta})}{n} \tag{47}$$

and then, by plugging Equation (47) into the log-likelihood function in Equation (46) and eliminating any constant terms, the MLE becomes a problem of solving the following minimization:

$$\underset{\theta}{\operatorname{argmax}}\,(\ln L) = \underset{\theta}{\operatorname{argmin}} \left(\frac{n \ln \hat{\sigma}^2 - \ln |\boldsymbol{R}|}{2} \right) \tag{48}$$

or

$$\underset{\theta}{\operatorname{argmin}} \left(\hat{\sigma}^2 \, |\boldsymbol{R}|^{1/n} \right) \tag{49}$$

This minimization problem, of course, does not have a closed-form solution, so a nonlinear iterative solver must be used to find the optimal value of the parameter vector θ.

As aforementioned, the origin of DACE method is Kriging method in geostatistics, where the data acquisition is often very expensive. With no doubt, Kriging method has naturally been evolved to provide very good prediction performance even with a small number of samples, and so thus the DACE method, the derivative of Kriging method. The DACE method also provides the model's full adaptability to the data containing high-order nonlinearity, especially multi-modal responses, so that the model fitting process can be done easily without requiring users a prior knowledge of the model behaviors. Another key characteristic of the DACE model is that all provided sample responses are exactly fitted, but it also means that the DACE model may have a weak ability to fit for the data contaminated with random errors.

There are of course shortcomings of the DACE method. For the training data of the size n, a DACE model requires the computation of the inverse matrix of $\boldsymbol{R}_{n\times n}$ and the correlation vector $r(x)$ having n correlation functions (which are usually Gaussian-type exponential functions). When it comes to the training data with a large number of samples, the massive mathematical expression of the DACE model causes significant computational burdens in both the training process and actual use of the model for prediction or simulation. As explained beforehand, the training process is a nonlinear optimization problem which requires the iterative solving approach, and each iteration needs the inversion of $\boldsymbol{R}$, the $n \times n$ correlation matrix which makes the use of DACE modeling prohibitive even with the data with several thousands of samples.

Recalling that one of the main reason for using a surrogate model is to take an advantage of its cheaper computation cost, The Kriging model will no longer provide such a merit if a large number of samples are needed or preferred. However, the Kriging modeling method should be a very appealing approach especially for the application of computer experiments if a small size of data is imposed, along with its adaptability, nonlinear capability, and deterministic interpolation capability.

The applications of Kriging modeling have been mostly for generating static models. Although very few, there are efforts of applying the Kriging or Gaussian process modeling method to dynamic modeling: Wang et al. [83] developed Gaussian Process Dynamic Model (GPDM) method and used it for modeling human motion which inherently needs a large-dimensional state space with the use of a small data set; Kocijan et al. [36] introduced the identification of dynamic systems with Gaussian process modeling. Although there are some additions of probabilistic or mathematical tools for dynamic system modeling, those approaches were not significantly different from the method for static models introduced above. The only significant difference is that the discrete time delays of the system outputs were used as the part of the input vector x. As a result, their approach for dynamic modeling has the same benefits and drawback of the static modeling approach explained above.

After all, as the pros and cons of the DACE or Kriging model as the surrogate model structure for dynamic system components are summarized in Table reftab:dace.

Table 4: Pros and Cons of DACE Model

Pros	Cons
• Very good prediction with a small sample set. • Nonlinear mapping performance. • No need for a prior knowledge of the system response or system. • Exactly fit on sample points.	• Not many examples of dynamic modeling cases. • n correlation functions for n size data. • Iterative training requires the inverse of matrix $\boldsymbol{R}_{n\times n}$ at each iteration.

2.3.3 Basis Function Expansion Models

The other very common approach is to use the basis-function expansion as the model of a nonlinear system. In general, a parametric estimation model of a nonlinear dynamic system can be expressed with a nonlinear mapping $G(\cdot)$ such as

$$\hat{y}(t|\theta) = G(\varphi(t), \theta) \tag{50}$$

The nonlinear function $G(\cdot)$ can be approximated by the linear combination of a sufficient number of (ideally) orthogonal bases in the function space for $G(\cdot)$. This idea can be described as

$$G(\varphi(t), \theta) = \sum_{i}^{n} \alpha_i g_i(\varphi(t), \beta_i, \gamma_i) \tag{51}$$

where g_i is a basis function, and the parameters α_i, β_i, and γ_i are the subset of the model parameters θ. Among them, α_i is a scalar, but β_i and γ_i can be either scalars or vectors.

A common mathematical form of g_i is called a mother basis function, denoted by $\kappa(\cdot)$, from which different functions g_i are created based on the variation of the parameters β_i and γ_i. Specifically in a basis function, β_i is referred to as the dilation parameters, and γ_i as the translation parameters.

Based on how the regressors, dilation parameters, and translation parameters are related in g_i, there are three different types of mother basis functions, which are tensor product, radial construction, and ridge construction [41, pp.150-151].

Assuming the regression vector $\varphi \in \mathbb{R}^m$, the *tensor product* forms a basis function g_i in the following way,

$$g_i(\varphi, \beta_i, \gamma_i) = \prod_{j=1}^{m} \kappa(\beta_{ij}(\varphi_j - \gamma_{ij})) \tag{52}$$

where β_{ij} and γ_{ij} are the elements of the vectors β_i, $\gamma_i \in \mathbb{R}^m$, with $j = 1, 2, \dots, m$.

The *ridge construction* of a basis function is somewhat simpler than the tensor product, that is,

$$g_i(\varphi, \beta_i, \gamma_i) = \kappa(\beta_i^T \varphi - \gamma_i) \tag{53}$$

where $\beta_i \in \mathbb{R}^m$ and $\gamma_i \in \mathbb{R}^1$. Note that the mother basis function κ takes a scalar input that is basically generated by the inner product of the dilation parameters β_i and the regression vector φ. A good example of the basis-function model using the ridge construction is the famous single hidden-layer (SHL) feedforward NN with the sigmoid function as the mother basis function. In the field of ANN, the basis function is called an activation function.

Lastly, the *radial construction* is described by

$$g_i(\varphi, \beta_i, \gamma_i) = \kappa(\beta_i \left\| \varphi - \gamma_i \right\|) \tag{54}$$

where, this time, β_i is scalar and $\gamma_i \in \mathbb{R}^m$. The the basis function is formed by the Euclidean distance of the current values of the regressors $\varphi(t)$ from a given center point, γ_i. A well-known example based on the radial construction is the radial basis-function (RBF) networks and wavelet networks.

In various basis-function expansion models, the two especially popular neural network models – feedforward sigmoid networks and radial-basis function networks – will be introduced further.

2.3.3.1 Feedforward Neural Networks with Sigmoid Activation Function

Motivated by biological neural networks, the artificial neural networks were created from the field of artificial intelligence. In about three decades, NN have gained an overwhelming popularity over almost all kinds of engineering fields, including design and optimization, system identification, and controls too.

A simple scheme of the architecture of the feedforward neural net is in Figure 20. The simplest form of feadforward neural nets consists of three layers: the input layer is just a simple place holder of the model inputs; the hidden layer consists of multiple neurons (or nodes) that act as nonlinear mapping elements; and the output layer is where outputs are yielded. The most common choices as the activation function for the output layer are the linear function or linear saturated function but one can always select nonlinear functions for its own purpose.

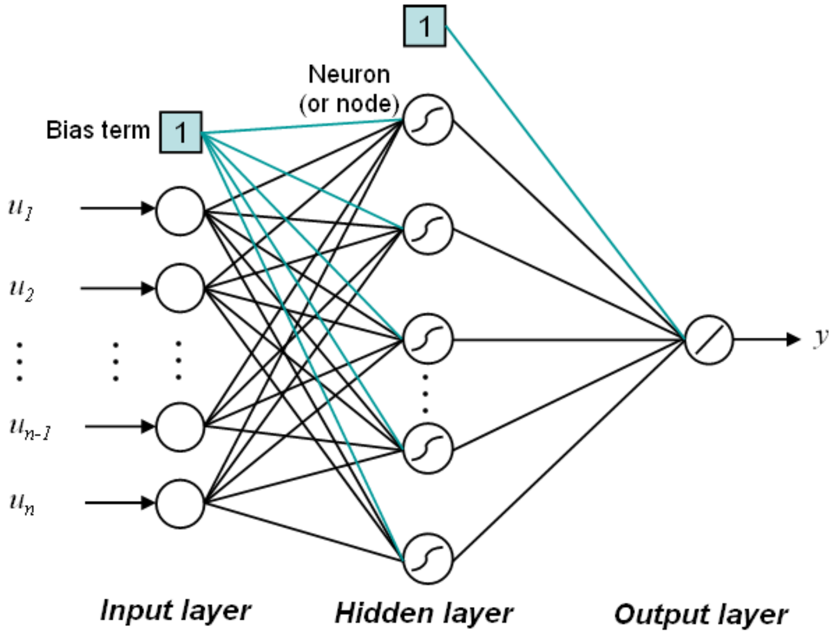


Figure 20: Single Hidden Layer Feedforward Neural Network with Single Output

Neurons (Figure 21) located in hidden and output layers are characterized by their activation function. In Figure 21, $f(x)$ is an activation function, w_i the weights of the inputs to the neuron, and b a bias. In terms of basis-function expansion modeling in

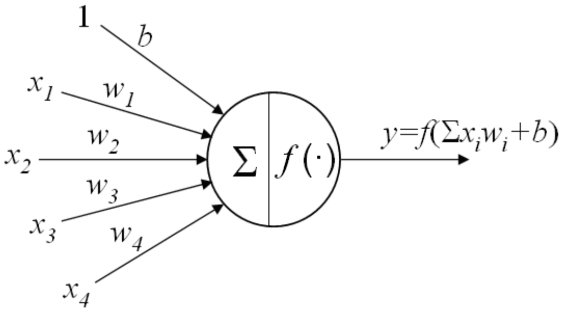


Figure 21: A Notional Neuron

§2.3.3, $f(x)$ is the basis function, w_i the elements of the dilation parameter vector β_i, and b the translation parameter γ_i. For the sigmoid activation function, the logistic function in Equation (55) and the hyperbolic tangent function in Equation (56) are two very common choices.

$$f(x) = \frac{1}{1+e^{-x}} \tag{55}$$

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{56}$$

Though the sigmoid neural net can have multiple hidden layers – this more general architecture is also known as the famous multi-layer perceptron (MLP), a single hidden layer (SHL) as shown in Figure 20 is popular and often enough for typical use since a SHL neural net with the sigmoid activation function has been proven to be able to approximate any smooth nonlinear system arbitrarily well [33], and are simpler and numerically lighter than multi hidden layer neural nets.

A training process of a neural net is referred to as a back-propagation process. A back-propagation process is basically a nonlinear optimization problem which finds the best estimates of the weights and the biases of the NN that minimizes the output error. A typical way of measuring the output error is the mean squared error (MSE). Since the training process is a nonlinear optimization problem and involved with a large dimension of variables (i.e., weights and biases), it tends to fall into local minima, so the training process is often performed multiple times with different initial conditions to have a better possibility to catch the global minimum or near-global minima. Though a neural net is a universal approximator in theory, its actual application is strongly limited by the complexity of the optimization problem, abilities of currently available back-propagation algorithms, and computational burdens.

One of the other drawbacks of NN is that the structural properties of a NN such as the number of hidden layers and nodes in each hidden layer should be, in general, determined by experimental, trial-and-error approaches. As a result, a neural net in use does not necessarily have an optimal structure. In order to address this issue, several constructive learning algorithms have been developed largely in two completely opposite approaches, one of which is network-growing, whose example is the cascade-correlation learning algorithm [24], and another is network-pruning which includes the optimal brain damage and the optimal brain surgeon algorithms [63, pp.221-237].

2.3.3.2 Radial-Basis Function Networks

The radial-basis function (RBF) network is recognized as a variant of neural networks by many people but is, in fact, an older technology than neural nets and originated from the conventional approximation theory, not the A.I. communities [20].

The architecture of the RBF nets is basically identical with that of the feedforward NNs, except for its radial basis functions for the nodes of the hidden layers and the absence of bias terms in the hidden nodes, as shown in Figure 22. Contrary to MLP which allows multiple hidden layers in their structures, RBF-net only allows the single hidden-layer structure.

A Gaussian function is one of commonly used radial-basis forms, that is

$$f(x) = \exp\left(-\frac{\|x-c\|^2}{2\sigma^2}\right) \tag{57}$$

where, x and $c \in \mathbb{R}^m$, c is called the center point, and σ the smoothness parameter

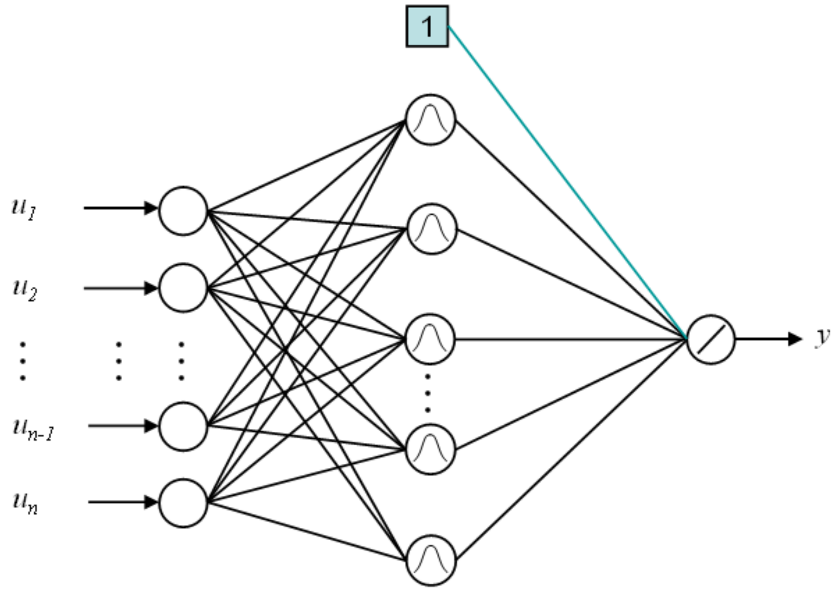


Figure 22: Radial-Basis Function Networks with Single Output

of the node. σ is used to adjust the influence of this node in the prediction of the response at a point x. The Gaussian radial-basis function is very similar to the correlation function of the DACE method in Equation (31), in §2.3.2, where the parameters θ and w do the same role of $1/2\sigma^2$ and c of the radial-basis function in Equation (57).

According to the general notations of basis-function expansions in §2.3.3, the parameters c and σ can be referred to as the translation and the dilation parameters of the radial-basis function. There are many other types of RBF of which several examples can be found in [20], but in fact, any function $f(\cdot)$ can be chosen as a RBF if it is continuous and monotonously convergent to 0 as $x \rightarrow \infty$.

The training process of RBF nets is quite different with that of MLPs. It can be separated into two serial tasks. First, the determination of center points c_j and smoothness parameters σ (or σ_j for each node) are determined for the nodes of the hidden layer, where $j = 1, \ldots, N_{HL}$ is the index of the nodes in the hidden layer. Then, the weights w_{ij} of the RBF outputs to the output layer is estimated, where

Table 5: Pros and Cons of Sigmoid-NN and RBF-Net Model

Pros		**Cons**	
• Proven nonlinear mapping capability. • Many NN tools available. • Many well-developed training methods and frameworks available.		• Not for simulation (NOE) modeling. • Requires a relatively large data set.	
Sigmoid net	**RBFN**	**Sigmoid net**	**RBFN**
	• DACE-like behavior when there are n neurons for n data points. • Training provides global minimum.	• Over-fit problem. • No. of layers and neurons found by trial-and-error. • Training easily fall into local minimum.	• Complicated data-clustering analysis required for training RBF-net.

$i = 1, \ldots, N_{OL}$, $j = 1, \ldots, N_{HL}$, and N_{OL} is the number of nodes in the output layer. The first task is performed either based on heuristic rules or data-clustering analyses. The detailed approaches of those are well explained in Du and Swamy [20]. The second task can just be performed with any linear least squares method.

The pros and cons of sigmoid-net and RBF-net are summarized and compared in Table 5, based on the investigation of the two basis-function models. Regarding the consideration of which network is a better choice between MLP and RBF nets, Du and Swamy again provides a good demonstration result of the comparison MLP and RBF nets. In its example study of beam-forming modeling of an antenna [20, pp.291-292], RBF nets performed better in training speed and fitting performance, but MLP was better in the generalization performance and the model speed in simulation.

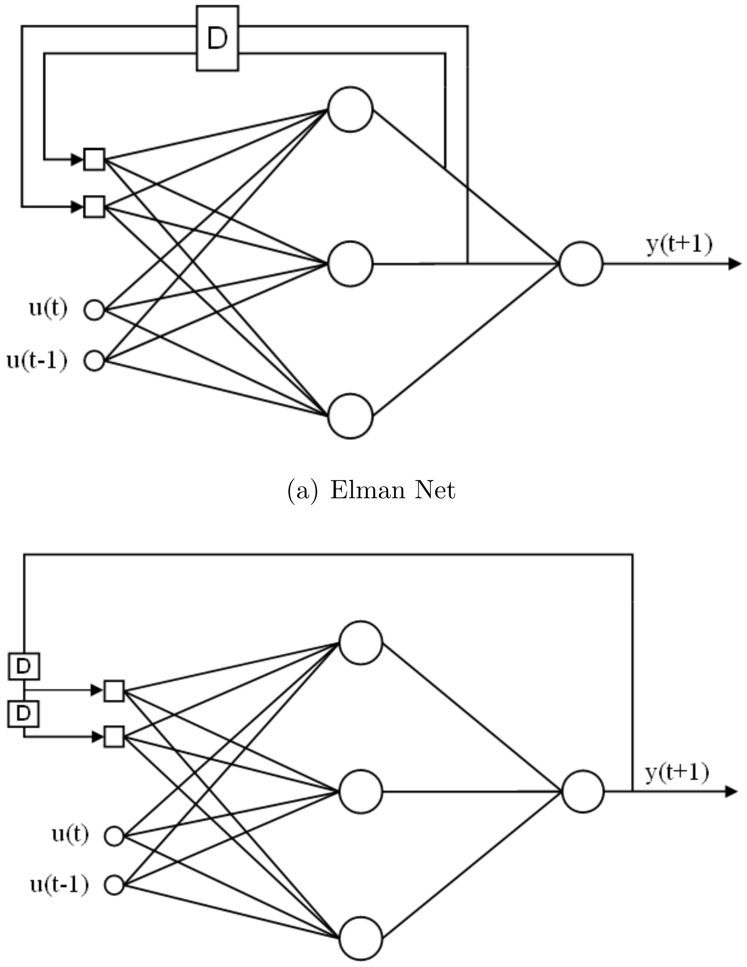
2.3.3.3 Recurrent Neural Networks

The architectures of the previously discussed neural nets are suitable for modeling static systems and some types of dynamic systems; however, they are not applicable to NOE-type modeling, which is most preferred in the dynamic-system simulation environment. The recurrent neural network (RNN) [45, 63], a variant of neural networks for the purpose of modeling of dynamic discrete systems and time-series forecasting, addresses this problem with its structural capability of storing long-term memory of input-output dynamics in their structure. The main difference between RNN and the static, memoryless neural networks in the previous section is the addition of time-lagged feedback connections in the network architecture. Otherwise, RNNs share almost the same architecture with the static feedforward NNs.

Depending on where the feedback branches out, a recurrent neural network is categorized as either an Elman network or a Jordan network. In an Elman network, the outputs of hidden layers are fed back to the input layer. In a Jordan network, it is the outputs of the network (i.e., the output layer) that are fed back to the input layer. Figure 23 shows the simplified structures of both Elman and Jordan networks. For dynamic surrogate modeling, a Jordan net may be a better choice because its feedback structure is more suitable for simulation during which the outputs of the model are supposed to be fed back anyway.

The identification of Elman nets requires recursive learning algorithms because the feedbacks from the outputs of the neurons in a hidden layer are unknown until the model is run and vary with the change of the weights throughout the training process.

For Jordan nets, usual batch-mode learning algorithms, which are the same algorithms used for the static feedforward neural nets, as well as recursive algorithms can be used for the identification, with appropriate preprocessing of the the training set. However, the identified model using a batch-mode approach – also called the



(a) Elman Net

(b) Jordan Net

Figure 23: Recurrent Neural Networks (For simplicity of description, bias terms are omitted, and SISO is assumed.)

series-parallel approach – is less robust in simulation uses than the model obtained by recursive identification. Anyway, this feature of a Jordan net can be very advantageous especially when the size of the training data set is large, since the recursive algorithms require significantly more computational burdens than the batch learning algorithms as a large size of training data.

Some commonly used recursive algorithms for training RNN are backpropagation through time algorithm (BPTT) by Werbos [85], real-time recurrent learning (RTRL) by Williams and Zipser [87], and extended recursive least-squares algorithm (ERLS) by Baltersee and Chambers [12].

Neural nets seem well suited to model nonlinear, dynamic black-box systems since they are theoretically able to adapt themselves to model qualitative and quantitative behaviors of the original systems by learning directly from the given training data, instead of depending on a priori information of the systems. It is mainly possible because of their highly general mathematical structures which are characterized only by parameters within them.

However the structural generality also becomes one of the weaknesses of neural nets. With the absence of any structural a priori information about the system process, an identified NN model often suffers from the problems of local over-fit and uneven error distribution of the model output, and it is hard to identify and control those problems when generating an NN model. For static modeling, managing model accuracy with a certain static error criterion representing the model's output accuracy such as model fit error (MFE) and model representation error (MRE) is good enough for obtaining a model with a good accuracy, but this approach is not suitable in the case of dynamic system modeling approaches. As a part of the inputs, the RNN model has the output feedbacks which also contain the output errors from the model at each simulation time step. Since a simulation is proceeded with the successive feedbacks of the model output, the error contained in the model output is also fed and propagates successively. This dynamic characteristics of error propagation, or the error dynamics of the model, can affect the stability of the simulation significantly, and unfortunately, dynamic black-box models with generalized model structures such as the RNN models have considerably higher tendency to have model instability in the simulation use than the structured models such as linear and polynomial models, even if the model maintains reasonably small MRE or MFE. Furthermore, there seem no good literatures and studies about predicting and controlling the error dynamics of dynamic black-box models with generalized model structures. As a result, the identification of a neural net-based dynamic surrogate model often requires a large size of training data obtained from an extensive and rigorous set of computer experiments in order to ensure the robustness, reliability, and stability of the model, but in some case even such an effort seldom help a neural-net model overcome the poor modeling performance due to the deficiency of its model structure.

2.3.4 Wiener Model: A Block-Oriented Nonlinear Parametric Model

As mentioned in §2.3.1, there are two main problems in the Volterra model. The first problem is that the identification of the Volterra kernel is difficult because the terms of Volterra series do not generate orthogonal outputs, and the second problem is that the model response converges only with the limited ranges of the input space. In order to address the problems, Wiener developed an alternative form of the Volterra expansion. This new form of expansion is called the non-homogenenous G-functionals, which creates orthogonal homogeneous Volterra functionals, under the assumption that the system was excited by Gaussian white noises and the kernels were expanded by an orthonormal function decomposition. A further introduction of the alternative form of the Volterra representation, which is called the generalized Wiener model representation is omitted here because of the complexity and the large volume of its theoretical bases and mathematical derivations, but instead one can find its details from Schetzen [67] and Ogunfunmi [57].

If the terms (which are called homogeneous G-functionals) of the Wiener representation are further expanded using linear orthogonal bases such as Laguerre series, the Wiener model reveals an interesting structure for nonlinear time-invariant systems modeling. Using the Wiener model representation with the nonlinear order N, a system model can be expressed by

$$y_N(t) = \sum_{p=0}^{N} G_p\left[k_p; u(t)\right] \tag{58}$$

where G_p is the pth order homogeneous G-functional with the Wiener kernel k_p, which is expressed as

$$G_p\left[k_p;u(t)\right]=\int_0^\infty\dots\int_0^\infty k_p(\tau_1,\cdots,\tau_p)u(t-\tau_1)\dots u(t-\tau_p)\;d\tau_1\dots d\tau_p \tag{59}$$

The pth order Wiener kernel can be expanded by the orthogonal bases such as Laguerre [67, 79, 80] functions, so the kernel $k_p(\cdot)$ in Equation (59) is now expressed as

$$k_p(\tau_1,\cdots,\tau_p)=\sum_{n_1=0}^{\infty}\cdots\sum_{n_p=0}^{\infty}c_{n_1\cdots\,n_p}\;l_{n_1}(\tau_1)\cdots l_{n_p}(\tau_p) \tag{60}$$

Then, by substituting Equations (59) and (60) to Equation (58), the model will be

$$\begin{aligned}
y_N(t)\;&=\;k_0+\sum_{n_1=0}^{\infty}c_{n_1}\int_0^\infty l_{n_1}(\tau_1)u(t-\tau_1)\;d\tau_1\\
&+\;\sum_{n_1=0}^{\infty}\sum_{n_2=0}^{\infty}c_{n_1n_2}\int_0^\infty\int_0^\infty l_{n_1}(\tau_1)l_{n_2}(\tau_2)u(t-\tau_1)u(t-\tau_2)\;d\tau_1d\tau_2\\
&+\;\cdots\\
&+\;\sum_{n_1=0}^{\infty}\cdots\sum_{n_p=0}^{\infty}c_{n_1\cdots n_p}\int_0^\infty\cdots\int_0^\infty l_{n_1}(\tau_1)\cdots l_{n_p}(\tau_p)\dots\\
&\quad\;\dots u(t-\tau_1)\cdots u(t-\tau_p)\;d\tau_1\cdots d\tau_p+\cdots\\
&+\;\sum_{n_1=0}^{\infty}\cdots\sum_{n_N=0}^{\infty}c_{n_1\cdots n_N}\int_0^\infty\cdots\int_0^\infty l_{n_1}(\tau_1)\cdots l_{n_N}(\tau_N)\dots\\
&\quad\;\dots u(t-\tau_1)\cdots u(t-\tau_N)\;d\tau_1\cdots d\tau_N
\end{aligned} \tag{61}$$

By letting $z_n(t)$ be the output of the Laguerre filter l_n, that is,

$$z_n(t)=\int_0^\infty l_n(\tau)u(t-\tau)d\tau \tag{62}$$

Finally, using the notation in Equation (62), the Wiener model equation in (61) can be expressed in the form of Equation (63):

$$\begin{aligned}
y_N(t) &= k_0 + \sum_{n_1=0}^{\infty} c_{n_1} z_{n_1}(t) \\
&+ \sum_{n_1=0}^{\infty} \sum_{n_2=0}^{\infty} c_{n_1 n_2} z_{n_1}(t) z_{n_2}(t) \\
&+ \cdots \\
&+ \sum_{n_1=0}^{\infty} \cdots \sum_{n_p=0}^{\infty} c_{n_1 \cdots n_p} z_{n_1}(t) \cdots z_{n_p}(t) \\
&+ \cdots \\
&+ \sum_{n_1=0}^{\infty} \cdots \sum_{n_N=0}^{\infty} c_{n_1 \cdots n_N} z_{n_1}(t) \cdots z_{n_N}(t) \qquad (63)
\end{aligned}$$

The Wiener model representation in Equation (63) tells important information about nonlinear system modeling, which is, the system can be approximated by the combination of a parallel set of orthogonal linear models with memory (i.e., linear dynamic models) in the form of Laguerre filters and a static nonlinear mapping from the outputs of the linear models $z_p(t)$ (with $p = 1, \ldots, N$) to the system output $y_p(t)$, as shown in Figure 24.

A significant number of researchers in the field of systems modeling and identification have expressed considerable interest with the Wiener model structure, because they can apply the well-developed linear theories for identifying or analyzing nonlinear systems. Nowadays, general models that have the structure of a linear dynamic block with a nonlinear static output mapping, or the systems with the same structural process are referred to as Wiener models, or Wiener systems, regardless of the orthogonality considerations in their formulations.

Many modeling approaches have been developed based on this typified structure. For instance, Hagenblad et al. [32] used an OE (output error)-type discrete-time difference equation model as the linear block with the polynomial fit model as the static nonlinear block, Janczak [34] also applied the OE-type difference equation model for the linear block and MLP for the static nonlinear part, Westwick and

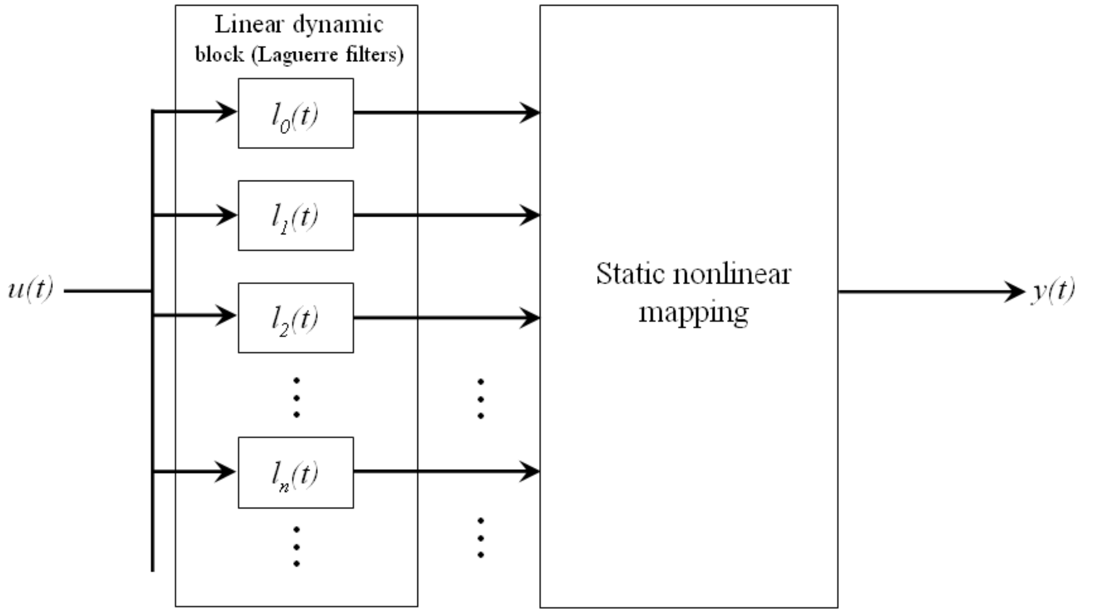


Figure 24: Structure of Wiener Model

Verhaegen [86] used MOESP (Multivariable Output-Error State-sPace) model as the linear block and polynomial mapping as the nonlinear block, and Al-Duwaish et al. [9] generated the linear model from the linearization of the nonlinear system at a certain operating condition and applied NN for nonlinear static mapping. A common characteristic of those approaches is that the linear block has the state feedback as shown in Figure 25.

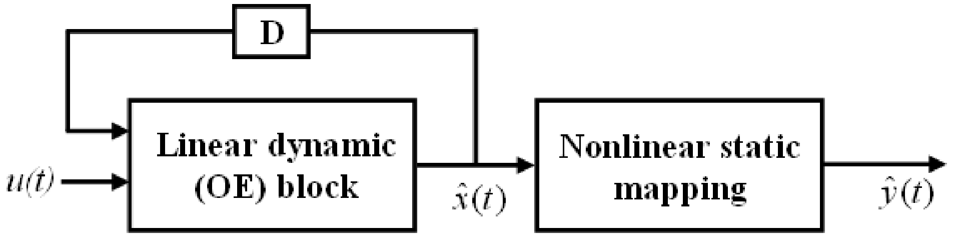


Figure 25: SISO Wiener Model with Feedback Linear Block

As briefly mentioned before, the models in the previous applications only adopted the block-oriented structure of the Wiener model and were not formulated or derived based on the mathematical formulation of the Wiener expansion in Equations (61) and (63). It means that those models are not guaranteed to work for general nonlinear time-invariant systems as the Volterra and Wiener models are. In fact, one of the assumptions in their formulations and applications was that the system to be identified also had the same block structure in its process, which limits their uses for nonlinear system modeling and yields relatively low model accuracy when they are applied to nonlinear models without a similar process structure.

When it comes to black-box modeling, the Wiener model in Figure 25 introduces a new state variable $x(t)$, which is not measurable from its original system, and its estimate $\hat{x}(t)$. Since this state variable is unknown during an identification process, a recursive-type estimation algorithm, which is numerically more expensive than a batch-mode algorithm, should be applied for the model identification. For simulation, the initial values of its delayed feedbacks of $\hat{x}(t)$ should be given beforehand. Since those values are unknown, there should be a process for estimating the initial values before the simulation, which can be a cumbersome task. Instead of the process for the estimation of initial values, there is an approach of neglecting the simulation outputs

Table 6: Pros and Cons of Wiener Model

Pros	**Cons**
• Capable of modeling nonlinearity in some level. • Can apply well-developed linear theories. • Dynamic characteristics are determined by its linear dynamic block. • Great model stability and robustness by the linear dynamic block.	• Valid only for modeling a system with Wiener structure, or model accuracy will be very low. • For OE linear model, a recursive model identification algorithm is required. It is numerically more expensive than batch-mode algorithms • For OE linear model, a cumbersome process of identifying the initial value of $\hat{x}(t)$ is needed.

from the first a few time steps.

There are other choices that do not introduces feedbacks in the linear dynamic block such as FIR models [31] or Laguerre filters [79, 80, 8] like the traditional Wiener model formulation. In those cases, the simulation can only be run in a fixed initial condition, which is $x_0 = 0$, as opposed to the model with a feedback linear block that can have an arbitrary initial condition for simulation. There is another drawback when a FIR model is as the linear block, which is, a FIR model requires a large number of input delays as regressors of the system model for good model accuracy. A larger dimension of the regression vector means a more computational burden in the identification of the model. More importantly in this research, it also means the increased complexity of managing interfaces between component models and the large size of data to be kept in the simulation of the aggregated system model. Lastly, based on the investigation of the block-oriented Wiener model in this section, the pros and cons are summarized in Table 6.

2.3.5 Other Block-Oriented Models

There are various other models based on the block-oriented approaches as well as the Wiener model, and one of them is the Hammerstein model [28, 11]. In the structural point of view, the Hammerstein model is simply the opposite of the Wiener model, so the system inputs are fed into a static nonlinear block first and then a linear dynamic block in sequence, as shown in Figure 26.

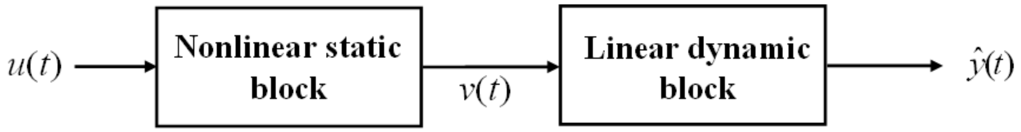


Figure 26: SISO Hammerstein Model

Another popular model is the Hammerstein-Wiener model. As the name implies, its structure is characterized by the concatenation of the Hammerstein and the Wiener models. Figure 27 is the simplified diagram of the Hammerstein-Wiener model.

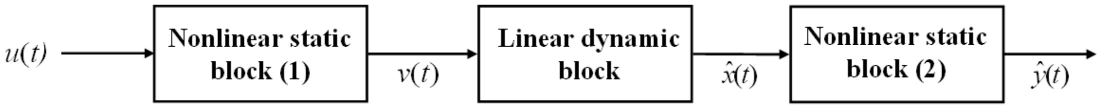


Figure 27: SISO Hammerstein-Wiener Model

CHAPTER III

SURROGATE MODELING FOR DYNAMIC NONLINEAR SYSTEM COMPONENTS

3.1 Introduction

The goal of this chapter is to formulate a surrogate modeling method for the component models of a large-scale ship fluid network whose behaviors are nonlinear and dynamic. The main focus in the development of the surrogate modeling method is on achieving or providing: 1) a sufficient level of fidelity in both quantitative and qualitative characteristics of the nonlinear, dynamic behaviors reliably; 2) model parsimony, which can be translated as the computational efficiency of the resultant surrogate model; 3) an efficient model-generation process by minimalizing the size of the training data needed and reducing subjective or expert interventions through the process so that a streamlined and largely automated surrogate modeling process is established.

In order to achieve these three objectives in the development of the surrogate modeling method, the following steps are proceeded as the development approach:

1. Design of surrogate model structure.
    (a) Selection of baseline model structure.
    (b) Design of model structure.
    (c) Design of regression vector.
2. Formulation of surrogate modeling process.
3. Example study (validation)

The key for achieving the previous objectives is directly or indirectly linked to the development of a good surrogate model structure. As the first step of building a model structure, a baseline model structure is selected from the nonlinear model structures that are investigated in Chapter 2. Since this baseline model structure may not satisfy some of the goals that were defined, the further modification of the baseline is performed in order to obtain a model structure that satisfies the goals better, and this modification also needs the design of the regression vector which defines the input of the surrogate model with the designed model structure. Then the improved model structure is used as a common surrogate model structure for the generation of the models of the fluid-system components.

Then, the process of generating a component surrogate model is developed. As mentioned at the beginning of Chapter 2, this process is similar to a typical process of static surrogate modeling in outline, but some task-level modifications are necessary to customize the surrogate modeling process to be suitable for dynamic systems applications, the new model structure, and the regression vector of the model structure.

Finally, two simple examples are given as the validation of the developed surrogate modeling method. In each example, surrogate models are generated with both the baseline and the modified structures, and the modeling performance and the training efficiency of the developed surrogate modeling method are evaluated by the comparison to the surrogate models with the baseline structure.

3.2 Design of Surrogate Model Structure

As the first step, a baseline model structure is chosen among the nonlinear model structures that are investigated in Chapter 2. The selection is based on qualitative evaluations of the characteristics of the model structures with respect to the goals that are set for the development of the surrogate modeling method of dynamic system components.

3.2.1 Selection of Baseline Model Structure

With revisiting and summarizing the pros and cons of the nonlinear model structures in in Chapter 2, Figure 28 provides the relative performance, or capability ratings, given to the model structures for the different requirements stated in §3.1, and the brief reasons of scoring for those requirements follow. In the model structures listed in Figure 28, the sigmoid- and RBF-nets are set to have the Jordan-net structure; in other words, they have output feedbacks in their structures.

Modeling nonlinearity. All the models are scored high on modeling nonlinearity, except for the Wiener model, since the Wiener model has a limitation in its application for nonlinear system modeling. The Wiener model delivers acceptable accuracy, only when it is used for modeling a system with Wiener structure.

Modeling dynamic systems. All the models are scored high on this capability, except for the DACE model. The DACE model is scored moderate due to its few literatures regarding its applications to dynamic systems. Lack of abundant research examples in dynamic system modeling cases implies that the selection of the DACE model could come with the risk of encountering with unknown technical difficulties while applying to dynamic system modeling.

Model stability/robustness. In the case of dynamic system modeling, it is safe to say that the reliability of a certain surrogate modeling approach can be represented by its model stability when the resulting model is used in simulation. Here in the thesis, the model stability of a surrogate model refers to the surrogate model's ability to maintain its model output error from the response of its original model within a certain reasonable tolerance ε in simulation. It is distinguished from system stability which is literally the stability of the actual system response with respect to time, in a certain region or space characterized by system inputs and the transitions of system states.

	Modeling nonlinearity	Modeling dynamic systems	Model stability/ robustness	Model parsimony	Training efficiency/ easiness	Process automation
Volterra model	●	●	⊙ (Divergence outside input ranges)	⊙	○	●
DACE model	●	⊙ (Few studies on dynamic problems)	⊙ (Few studies on OE type cases)	○	○	●
Sigmoid-RNN (Jordan)	●	●	⊙ (Too general model structure)	●	⊙ (Using batch-mode rules)	●
RBF-RNN (Jordan)	●	●	⊙ (Too general model structure)	⊙	⊙ (Using least squares rules)	●
Wiener model	⊙ (Only for Wiener structure)	●	● (Linear dynamic block)	N/A (Depends on model choices)	○	●

○	Bad
⊙	Moderate
●	Good

Figure 28: Comparison of Nonlinear Model Structures

In this sense, the Volterra model as a component surrogate model may not be a good choice because of its high vulnerability to model divergence when the model happens to operate outside input ranges covered by its training data. One may think that this divergence problem can be avoided by carefully setting the simulation environment, so the operation area of the simulation stays within the model's input ranges that would not cause the model divergence. Unfortunately, it is often very difficult in practice to know such a simulation operation area for all component surrogate models because of nonlinearity of the model response and complicated interactions of component surrogate models with others.

Unlike the Volterra model, the DACE model does not have the model divergence problem related to input ranges. the DACE model is, in the mathematical form, the linear combination of the Gaussian distribution functions, whose centers are located at the training data points. Since outputs of these Gaussian functions approach zero as their inputs go to infinity, the DACE model is strongly bounded and tends to be highly more stable than the Volterra model in the unexpected occasions of simulation's running outside the predesignated input ranges of the DACE model.

Similar to the DACE model, the both sigmoid- and RBF-net models are also bounded and highly stable to the simulation runs outside their input ranges, since the responses of their basis functions are all bounded. In fact, those RNN models still have problems of model instability from a different source which is, as mentioned in §2.3.3.3, the structural generality of the models, and this is also true to the DACE model when it is used with the ouput feedbacks on it as a simulation model since this model has a very generalized structure for nonlinear modeling too.

Model parsimony. The lowest score is given to the DACE model because it contains n Gaussian functions for the training data set with n samples. The DACE model will be very expensive for numerical implementation even with a few hundreds of samples for the training data set. Although it is very hard to find the subtle superiority in the performance of different models from this qualitative and subjective study, the RNN models seem better than Volterra model regarding model parsimony, especially when a large number of regressors are required. This is because the size of the NN model appears to be less likely coupled with the size of regression vector than that of the Volterra model. However, for a small amount of regressors, the Volterra model can be more affordable for numerical implementation due to its simple and efficient mathematical form, which is a polynomial function. Comparing between the two RNN models, the sigmoid-net is known to require a less number of neurons than the RBF-net for a given performance requirement [20]. Considering all these, the sigmoid-RNN is rated as good, and the RBF-net and Volterra models are rated as moderate. Rating the Wiener model is undetermined, since the model parsimony of it is dependent on which model forms are chosen for the linear dynamic and nonlinear static blocks of the model.

Training efficiency/easiness. As previously discussed in §ss:volterra and §2.3.2, the identification algorithm of the Volterra model is complicated and numerically expensive, and the iterative identification process of the DACE model is also numerically expensive. In the case of the Wiener model, the identification requires a recursive algorithm whose computational cost is very high. Considering the numerical efficiency of training a model and the easiness of implementing the training process, the RNN models are advantageous over the other three models. Another advantage of RNN models is that there are many free or commercial NN modeling framework tools available for both RNN models. Assuming the use of a batch-mode training algorithm for training sigmoid- and RBF-nets, most of the training algorithms are basically from the general-purpose nonlinear/linear solver or optimizer algorithms which are easy to understand and implement. For the RBF-net, the training process consists of two sub-processes: first, finding the parameters of the basis functions using data-clustering analysis, and second, identifying the coefficients for the linear combination of the basis functions, which can be performed with a simple least squares algorithm. Although the implementation of data-clustering process can be somewhat complicated, the training speed of an RBF-net is typically faster than that of a sigmoid-net trained using a simple training algorithm [20].

Process automation. There can be a certain model structure that is more efficient and easier for implementing automation of the training process than others, but basicall an automated training environment can be created with any of the investigated modeling structures, for the purpose of minimalizing the manual intervention for generating surrogate models.

Based on the brief review above, it is safe to say that none of the investigated model structures delivers all the desired capabilities for surrogate modeling of the fluid-system components. Nevertheless, the review also shows that the sigmoid RNN with output feedbacks may be the most reasonable and robust choice as the baseline model structure, based on the ratings. The sigmoid RNN still needs improvements in two capabilities, which are model stability and training efficiency, according to the ratings in Figure 28. Therefore, the baseline structure is modified to achieve the improvements in these two capabilities.

3.2.2 Choosing Fidelity of Transient Analysis: Transient Vs. Quasi-Steady State Simulation

Before building surrogate models of fluid model components, the expected level of fidelity on the transient modeling needs to be decided beforehand, because the approach for building models can differ depending on it. If the model is expected to provide relatively high accuracy of transient responses and details of fast dynamics, one must choose a dynamic surrogate model that contains some type of feedbacks for a long-term memory of the transient dynamics.

In the opposite case, one can build just a static surrogate model to perform quasi-steady state simulation which will be numerically lighter and easier to implement. An example of quasi-steady state approaches is based on the *extended-time simulation* of hydraulic systems [37]. In a simulation of a hydraulic system, the transient dynamics is often so fast that its influence to the system-level characteristics and performance can be considered negligible, when compared to its steady-state responses. In the extended-time simulation scheme, the system transient response at each time step is replaced by a steady state response at the current input values and system settings, assuming that the inertia effect is negligible. This type of simulation should provide good enough fidelity for some analyses for conceptual or preliminary design, although the detailed design, where the transient effects like the water hammering and inertia must become very important characteristics to know, will certainly need a high-fidelity transient analysis. In the thesis, a demand of dynamic surrogate modeling is assumed during the formulation of the model structure because it is obviously a more difficult problem to solve than well-developed static surrogate modeling approaches.

3.2.3 Recurrent Neural Network with Block-Oriented Structure

The formulation of the RNN with a block-oriented structure starts from a plain SHL RNN. As previously discussed in §2.3.3.3, the neural net has a very generalized mathematical structure which allows for modeling an arbitrary nonlinear system, but this generality of the structure often yields poor model stability and robustness in simulation use.

A remedy for such a drawback may be to impose a predefined structure onto the neural net model, like the Wiener structure in Figure 25. From one perspective, the Wiener model structure can be interpreted as a linear approximation model of the nonlinear system with an added transformation from linear estimates to local nonlinearity as a calibration of the estimates from the linear model. A linear identification model has a lower fidelity than a nonlinear model, but is inherently more robust in its predictions. Since the dynamic behavior of the whole model is predominantly determined by the linear block, a model with the Wiener structure takes advantage of the stability and reliability of the linear block, but is still capable of modeling nonlinearity.

The formulation of the Wiener-structured RNN can start from an OE-type linear model. Let us assume that the linear dynamic block model in Figure 25 is represented by an OE-type difference equation given as

$$\hat{x}(t) = f_1\hat{x}(t-1) + \cdots + f_{n_f}\hat{x}(t-n_f) + b_0u(t) + b_1u(t-1) + \cdots + b_{n_b}u(t-n_b) \quad (64)$$

The equivalent linear neural net can then be constructed as shown in Figure 29. By placing a SHL feedforward neural net as the static nonlinear block, the RNN with the Wiener structure can be formed as shown in Figure 30. This structured RNN is also an Elman net with two hidden layers. The NN with the Wiener structure can be identified with any recursive back-propagation algorithm built for general RNNs, which means one can take advantage of a number of well-developed neural net tools available. So far the approach is basically identical with Janczak [34], except that the

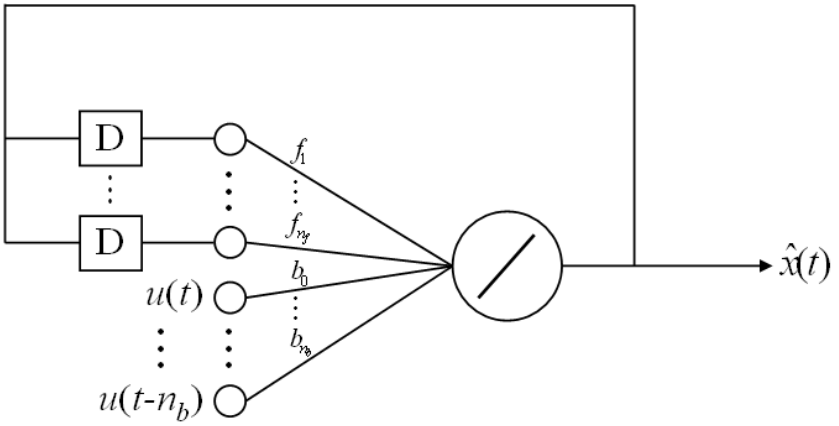


Figure 29: A Linear Neural Model as Linear Dynamic Block of Wiener Structure

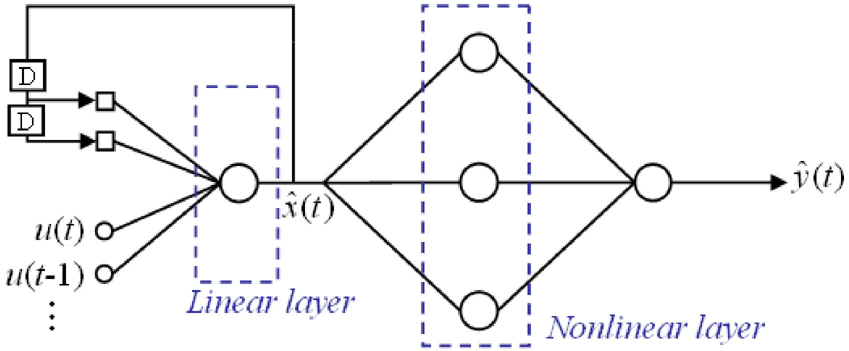


Figure 30: Double Hidden-Layer Neural Net with Wiener Structure

Janczak's linear model was not explicitly translated into the equivalent linear neural layer.

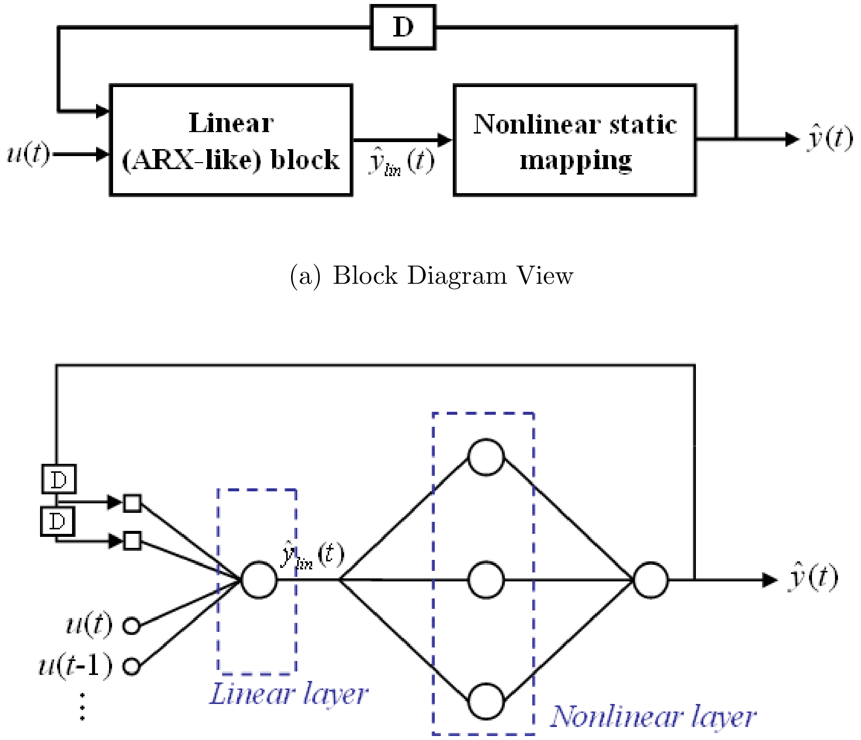
The drawbacks of the Wiener-NN are three-fold. First, only a time-consuming recursive learning algorithm is available for training such a neural-net structure. Second, the model accuracy is very poor in the case of nonlinear systems that do not have the Wiener structure. The benefit is that the model may provide the improvement in model stability. Third, because it is still a Wiener model, the initial values of the delayed state feedbacks must be estimated by a separate computational process whenever a simulation starts.

In order to address the drawbacks, a new modification is applied to the Wiener-NN, the idea being quite simple; just the hidden-layer feedbacks are replaced by the output feedbacks. The modified model structure is shown in Figure 31.

In the linear dynamic model, the difference equation is now expressed as

$$\hat{y}_{lin}(t) = a_1\hat{y}(t-1) + \cdots + n_a\hat{y}(t-n_a) + b_0u(t) + \cdots + b_{n_b}u(t-n_b) \qquad (65)$$

where $\hat{y}_{lin}(t)$ is the output estimate from the linear model, and $\hat{y}(t-i)$ is the delayed feedbacks of the output from the nonlinear block. From Equation (65), it can be observed that the linear model is more similar to an ARX model than an OE model since the model regression vector is not comprised of the delayed feedbacks of the



(a) Block Diagram View

(b) Equivalent Neural Net Representation

Figure 31: Block-Oriented Nonlinear Model with Output Feedback – the Modified Wiener Structure

output from itself, but the output from the static nonlinear block. This is supposed to provide an estimate closer to the real system response than the crude linear approximation from the linear model. Still, the overall model is an OE model because the inputs to the model are not from the tapped delay line of the true system response, but from the feedbacks of the simulation model. In other words, this block-oriented model is an NOE model of which the linear block has an ARX-like structure.

This modified block-oriented model can improve the three problems of the original Wiener model. As shown in Figure 31(b), the neural-net model with the modified-structure is now a Jordan net to which numerically more affordable batch learning algorithms can be applied in order to identify the model. In a Wiener model, the dynamic characteristics of the system are only modeled by the linear dynamic block.

In the model with the modified structure, the dynamic characteristics of the system are approximated by the blended efforts of both nonlinear and linear blocks. As a result, the model with the modified structure is expected to model a broader range of nonlinearities and have better accuracy than the original Wiener-structured model whose uses are somewhat limited to the systems with the Wiener structure. The modified structure does not contain the delayed state feedbacks of the linear dynamic block (i.e., the output of the linear block) so that the initial values of the delayed state feedbacks do not have to be estimated for simulation. Because the delayed output feedbacks are numerically more common and natural in simulation environments, the model with the modified structure is easier to use in simulations. Also the initial conditions are easy to set since they can be observed from the original system or model.

With all those improvements, the modified block structure retains the beneficial features of the Wiener structure. By applying the modified structure to the NN model, the NN model will have the better model stability and robustness by the ARX-like linear dynamic block. Because the model is realized in the NN, one can take advantage of the rich and matured resources of the NN generation methods and frameworks, and also well-developed numerical tools.

3.2.4 Design of Regression Vector

The purpose of surrogate modeling in this thesis is to enable the simulation-based design process. Thus the generated surrogate models should not be for the simulation of the system with only a single fixed design but also a group of design alternatives. However the formulation of the surrogate modeling method has been so far with only the consideration of simulating a system with a fixed design.

In order to realize the component surrogate models that can represent different component configurations, the regression vector should include model parameters.

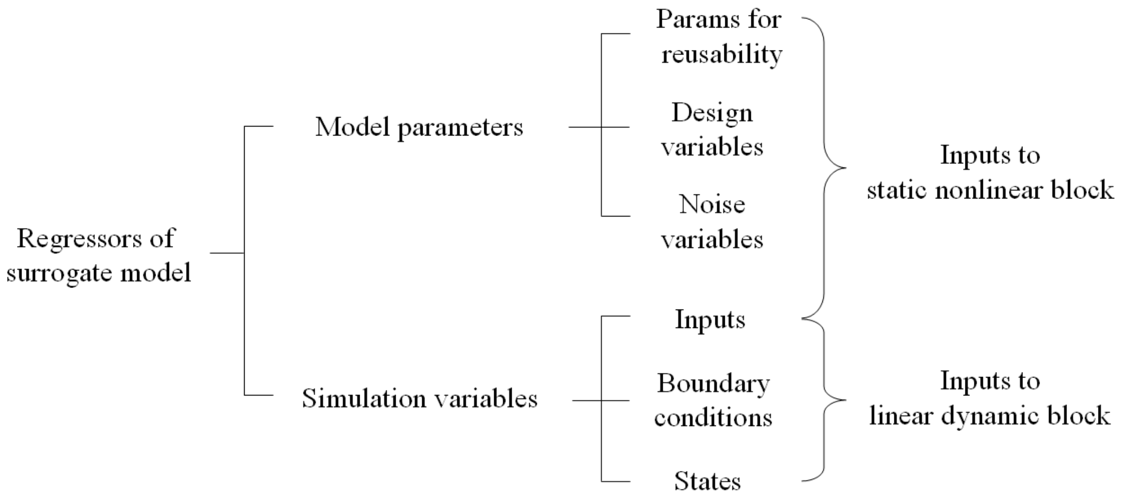


Figure 32: Classification of Variables as Regressors

Figure 32 is the classification of the variables that were used for dynamic surrogate modeling in this research.

The simulation variables vary only during the simulation run. They are for a surrogate model's communication with other component surrogate models, simulation scenarios, or controllers during a simulation, and the effect on the responses is mostly dynamic. In contrast, the parameters vary only between simulations for model reconfiguration. The change of the parameters also changes the overall system responses, which will be static for each simulation. In other words, the (functional) mapping from them to the system responses is static. Therefore, the variables affecting the dynamics of the responses will be referred to as the *dynamic variables*, and those changing the overall system responses statically (like changes of system design) will be referred to as the *static variables.*

As previously described, the proposed RNN model in §3.2.3 has an ARX-like linear dynamic layer and a static nonlinear layer. Reflecting the roles and characteristics of the dynamic and the static variables, the dynamic variables were set to be inputs to the linear dynamic layer, and the static variables to the static nonlinear layer.

There can be some simulation variable that is static even though it is used for simulation. An example can be found from the dynamics of the fluid flow at a valve: the functional relation between the flow rate and the valve opening ratio is dominantly static; but the flow rate and pressure difference at the two and of the system boundary has a dynamic relation. In that case, the pressure difference becomes the dynamic variable connected to the linear dynamic layer but the valve opening value should be the static variable connected to the static nonlinear layer even if it is one of simulation variables.

3.3 Generation of Surrogate Model

This section introduces a RNN-based surrogate modeling process that was formulated for the fluid model components. Figure 33 is the procedure of generating RNN surrogates, and the detailed explanation of each step follows in the subsections.

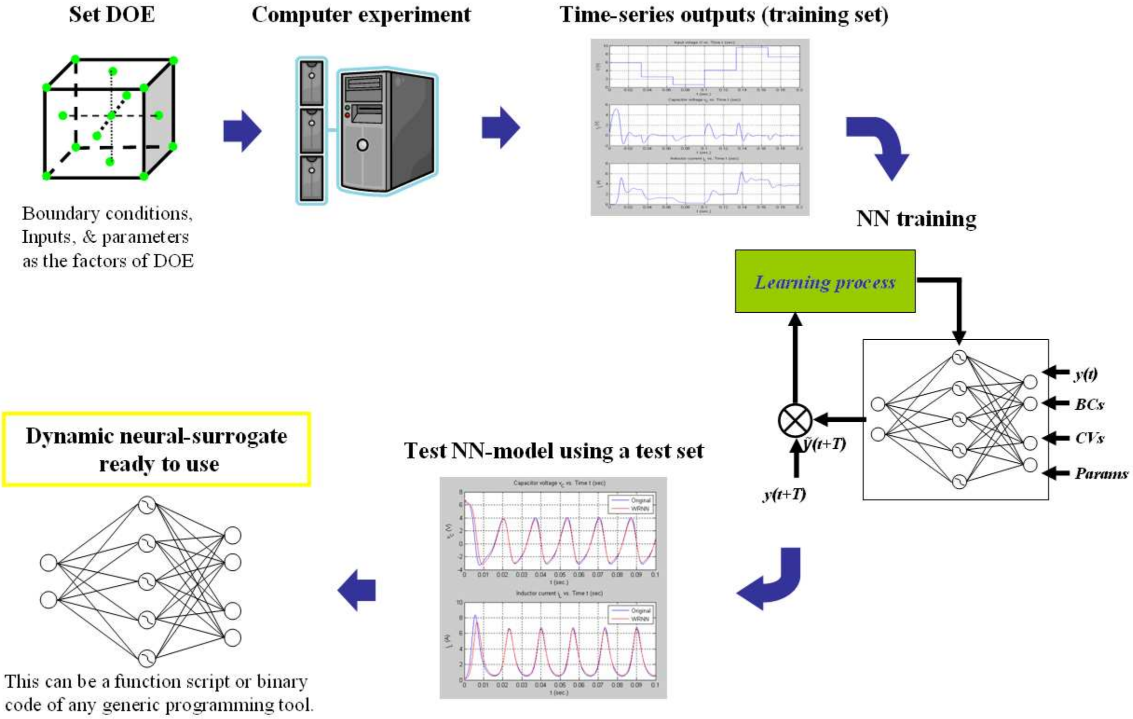

Set DOE
Computer experiment
Time-series outputs (training set)
Boundary conditions, Inputs, & parameters as the factors of DOE
NN training
Learning process
y(t)
BCs
CVs
Params
ŷ(t+T)
y(t+T)
Dynamic neural-surrogate ready to use
Test NN-model using a test set
Capacitor voltage v_C vs. Time t (sec)
Original
WRNN
t (sec.)
Inductor current i_L vs. Time t (sec)
This can be a function script or binary code of any generic programming tool.

Figure 33: RNN-Based Surrogate Modeling Process

3.3.1 Design of Experiments for Dynamic System Simulation

Design of Experiment (DOE) is “a systematic, rigorous approach to engineering problem-solving that applies principles and techniques at the data collection stage [7]” which is performed with a certain type of controlled experiments. In each experiment, the responses of a process, product, or system of interest is observed with a different set of the values of factors – the variables of which an experimenter studies the effect to the responses.

The use of DOE for static simulations is straightforward; the inputs are placed as the factors, the factor levels are defined, and then the DOE is created with the factors. However, when it comes to dynamic system simulations, such an implementation is not possible because neither the responses nor the factors' effects to the responses are static.

As pointed out in §3.2.4 and described in Figure 32, the approach of dynamic surrogate modeling yields not only static variables but also dynamic variables. Moreover, system state variables can not be part of the factors of DOE because they are uncontrollable during the simulation. As a result, the application of DOE for dynamic system simulations necessitates a different strategy than that for static problems.

The strategy is set in the following way: a DOE is generated with the static variables, and then another with the dynamic variables, excluding the state variables. Each experiment design in the first DOE array sets the static variables of the surrogate model, and a simulation is executed with the second DOE as the scheme for changing the dynamic variables, based on a time sequence. Therefore, the total number of simulation runs are determined by the first DOE, and the simulation time frame and scenario applied to every simulation run are commonly defined by the second DOE. This two stage-DOE strategy, which is also shown in Figure 34, happens to be similar to the technique used for building DOEs for robust parameter design [89]. The difference is that the second DOE is not for the simulation scenario for varying

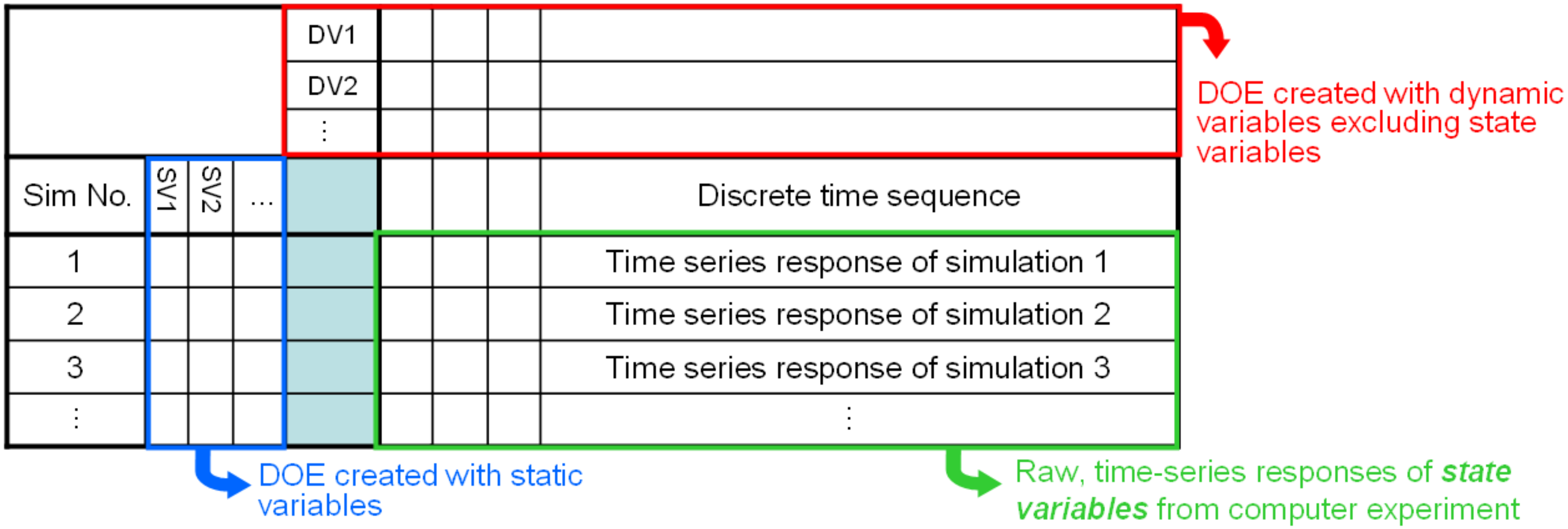

DV1
DV2
⋮
Sim No.
SV1
SV2
…
Discrete time sequence
1
2
3
⋮
Time series response of simulation 1
Time series response of simulation 2
Time series response of simulation 3
⋮
DOE created with dynamic variables excluding state variables
DOE created with static variables
Raw, time-series responses of ***state variables*** from computer experiment

Figure 34: Experimental strategy for dynamic system simulations

dynamic variables but for different settings of noise variables as the experiment's condition.

3.3.1.1 DOE Generation for Static and Dynamic Variables

For deterministic, computer simulation-based experiments, space-filling designs [64] are particularly popular. Examples of space-filling designs are maximum entropy design, maximum distance design, and Latin-hypercube design. As simpler alternatives that space-filling design, the full factorial design and the random sampling are also frequently used.

However, there are drawbacks in using space filling designs in some application. The space filling design algorithm alone does not construct the design points on the edges and corner points in the design space. Also, designers do not have the control of the sampling resolution of each factor, but instead, it is done by a DOE construction algorithm. Lastly, the algorithm involves with complicated mathematical and statistical processes so creating a DOE requires significant time and effort without proprietary application tools or numerical library.

In the case that the edges and corner points of the given design space are considered important to evaluate, one good option may be a hybrid design. An example of hybrid design is a space filling design combined with a low-resolution full factorial design that fills the corners, edges, and surfaces of the design space. However, this approach does not address the last two of the possible drawbacks of space filling designs.

As a simple option, the author has used a custom design by a simple modification of the full factorial design in similar applications [52]. This custom DOE is called two-stage modified factorial design by the author, and it requires a relatively less number of sample points than the full factorial design. A simple example of the two-factor modified factorial design with five factor levels on each factor is shown

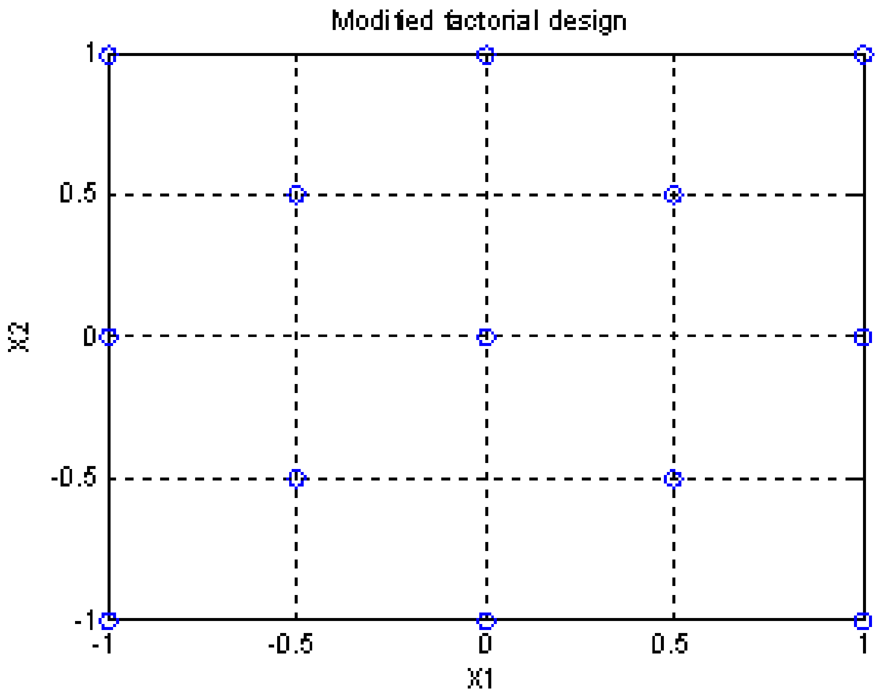


Figure 35: Modified Factorial Design, Two factors with Five Factor Levels

in Figure 35. The idea of the construction of this custom design is simple. A full factorial design is made with the odd levels of the factors. Then another full factorial design is made with the even levels and combined with the first one. The modified factorial design seems a good option to choose when the size of DOE and the three problems of space-filling designs matter.

3.3.1.2 Ranges and Sampling Resolutions of Variables

In order to perform a computer simulation based on DOE, the ranges of the factors and the resolution of the factor levels have to be determined. Unfortunately, their appropriate values are problem-specific, and have to be identified using trial-and-error approaches or engineer's intuition.

Logically, the DOE with narrower factor ranges is more advantageous in efficiency of model generation since fewer experiments will be needed than that with broader factor ranges for the same sampling density. However, assigning a very tight range can make the surrogate models unstable since there is a greater possibility of the simulation's running outside the range of the variables. A higher sampling resolution is also favorable for accuracy of a surrogate model but comes with the steep increase of computational cost.

3.3.2 Computer Experiment and Data Extraction

After the two-stage DOE and the ranges and resolutions of its factors are determined, computer simulations are run based on the DOE, and the raw data of the time-series responses from the simulations are obtained. The data for training neural nets are sampled out from the raw data. In a typical setting, the time-series response is the transient response of the system state variables, which are the outputs as well as the inputs of the surrogate model to be made. More specifically speaking, the current states are the function of the previous values of themselves, or auto-regressive.

3.3.2.1 Preprocessing of Simulation Results

In order to generate a neural surrogate model with a batch-mode training algorithm, the raw data should be preprocessed in a way that serializes the coupling of the outputs and inputs made by the recursion of the state variables. As an example, preprocessing of the data from the simulation of a SISO system is depicted in Figure 36. In the example, the number of the delayed feedbacks of the state $x(t)$ is two, and the tapped delays of the dynamic input $u_d(t)$ is just one. The model also has a static input u_s, either for the simulation, or as a static parameter.

In a batch-mode algorithm, the model is treated as a prediction model whose state values are all extracted from the data of the real system, even when it is actually a simulation (OE-type) model that takes the delayed outputs from itself. In a batch-mode training of a surrogate model, the true values of the state variables are accessible from the simulation result so the delayed state feedbacks to the surrogate model are tapped directly from it without having to execute the current surrogate model to

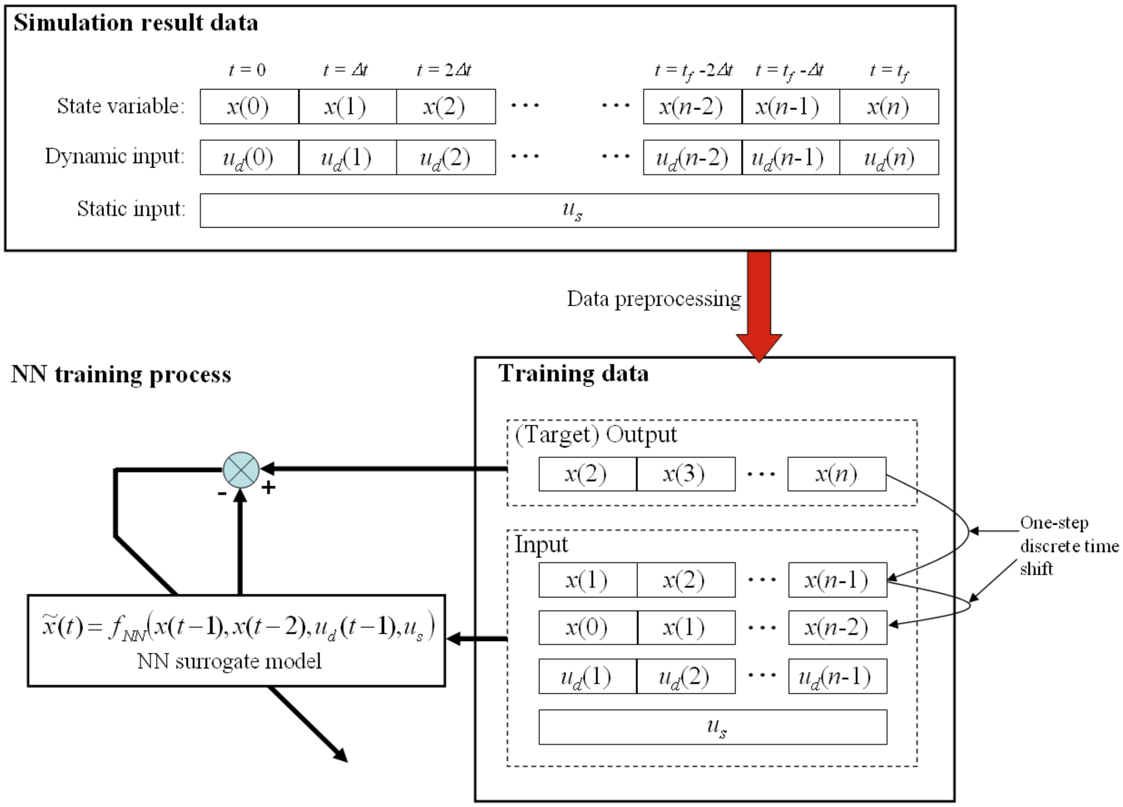


Figure 36: Preprocessing of Simulation Data for Batch-Mode Training of NN Surrogate Model

obtain the output estimates.

Since the array of the time-series history of the state variable is already given in a fixed time frame, the state variable array with a backward-shift of the discrete time step is equivalent to the array of the state feedback with a discrete-time delay. In Figure 36, the output (or target) part of the training data is occupied by the last $n - 1$ (from 2 to n) elements of the original array of state values. Then the input data includes the two $n - 1$ sized arrays of the state feedbacks, one with 1 to $n - 1$, and another with 0 to $n - 2$.

The processed training data can be used for any batch learning algorithm. During the training process, the neural net is just a feedforward NN that has tapped delays of the states as one of its inputs, although it is an RNN in actual simulation uses.

3.3.2.2 Data Normalization

When there are large differences in the order of magnitude among the variable values in the training data, it is strongly recommended that the training data be normalized. If such magnitudes are significantly different, the weights associated with the inputs with large magnitudes become abnormally larger than the rest, causing numerical difficulties in the learning process of the neural network [63]. Typically, the learning process of neural networks becomes more stable, yields more accurate mapping, and requires fewer iterations when the data are normalized [73]. There are several common methods for normalization, and in this research, the simple method of linearly transforming all the variables in the training data to the range of [-1, 1] was used.

3.3.3 Training, Testing, and Launching NN Surrogate Model

The performance of the neural network training depends strongly on the chosen backpropagation rule (i.e., training rule), thus the choice of a backpropagation rule is critical in setting up neural network training. As guidance, Seiffert [68] presented a good summary about the characteristics of different propagation rules, clarifying that the second-order training methods have exceptional performance compared to others. Among many available learning algorithms, Levenberg-Marquardt (LM) is one of the more popular choices since it requires less computation time and fewer training epochs, although this algorithm has the penalty of higher memory consumption.

In many neural-net applications for function approximation or modeling, there is a secondary data set called the *validation set*, which is used for preventing the neural net from over-fitting. For each iteration of training the neural net and evaluating the training errors using the training set, the validation error of the neural net is also evaluated using the validation set in order to measure the generalization performance of it. For an iteration, both the training and the validation errors are supposed to be decreased if over-fitting does not occur, but at some point of iteration, the validation error begins to increase while the training error still keep decreasing. Then the backpropagation algorithm considers it as the indication that over-fitting begins and stops the training process. This logic is referred to as *early stopping.*

Another way of avoiding the problem of over-fitting of a neural net is just using training data that is far larger than the degree of freedom of a neural net. In that case, using validation data is unnecessary.

Once a neural net is trained, it is tested with a fresh set of data called test data and finally used as a surrogate model. Since the mathematical form of a neural net model is relatively simple, it can be implemented with any scientific computing language making the model implementation very flexible to the user's modeling environment.

3.4 Example Study

In this example demonstration, the surrogate modeling method for fluid model components is tested and evaluated by applying it to two simple nonlinear systems, which are a nonlinear RLC circuit and a cooling-water pipeline with an heat exchanger and a flow control valve. For each example, two RNN models – one with the baseline structure, which is a simple SHL Jordan-type RNN, and another with the modified block structure proposed in §3.2 – are generated using the surrogate modeling process in §3.3. Then, the two surrogate models are compared with respect to model accuracy, stability, and training efficiency as the validation and demonstration of the benefits of the RNN-based surrogate modeling method with the modified block structure.

3.4.1 Nonlinear RLC Circuit

The first test model is a RLC-circuit with nonlinear inductance. The circuit layout is shown in Figure 37. The nonlinear inductor model is borrowed from the test model of Meliopoulos and Stefopoulos [50]. With the circuit layout in Figure 37, the system

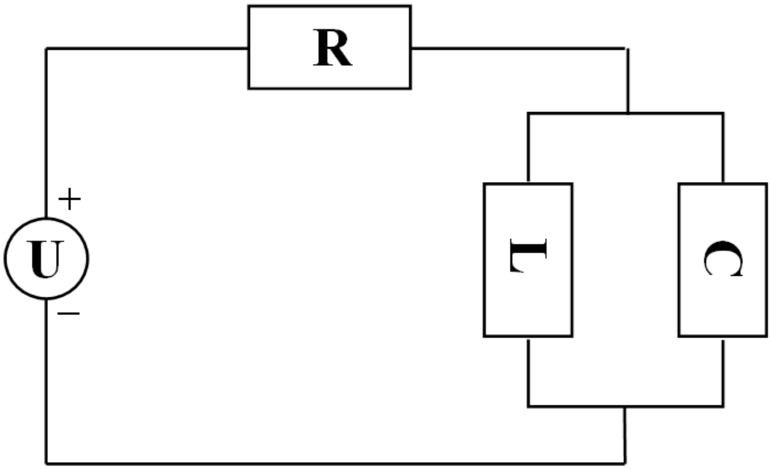


Figure 37: Simple RLC Circuit with Nonlinear Inductance

model equation is expressed as

$$\begin{aligned} \frac{d\lambda}{dt} &= v_C \\ \frac{dv_C}{dt} &= \frac{U}{RC} - \frac{v_C}{RC} - \frac{i_L}{C} \\ i_L &= i_{L_0} \left(\frac{\lambda}{\lambda_0} \right)^n sgn(\lambda) \end{aligned} \tag{66}$$

where λ is inductor flux, v_C is capacitor voltage, i_L is inductor current, and U is source voltage. λ_0 and i_{L_0} are the initial values of λ and i_L. The system parameters and initial values of the model are given in Table 7. The numerical RLC model and the two NN models were implemented into Matlab®.

Table 7: Model Settings of Nonlinear RLC Circuit

R (Ω)	C (F)	i_{L_0} (A)	λ_0 (Wb)	n
2	1.5×10^{-3}	10	0.07	8

3.4.1.1 Generation of NN Models

The configurations of the NN models with two different model structures are given in Table 8. The original system model has two state variables, which are the inductor current and the capacitor voltage in the RLC circuit, and these two states are set as the outputs of the NN surrogate models. In Table 8, the baseline SHL NN model is referred to as the plain NN model, whose hidden layer consists of 10 neurons with the hyperbolic tangent activation function. On the other hand, the block structured NN has two hidden layers, which are one linear layer and one nonlinear layer. The linear layer has two linear neurons, which are set to be the same amount as the state variables of the system. The nonlinear layer has 10 neurons with the hyperbolic tangent activation function, like the plain NN model. Comparing the configurations of the NN models in Table 8, the model with the modified block structure has two additional linear neurons over the the SHL NN model; however the block-structured NN model has a lower degree-of-freedom (the total number of model parameters such as weights and biases of an NN model) than the SHL NN model, due to its bottle-neck structure between the linear and nonlinear layers of the model.

Table 8: Configuration of Two NN Surrogate Model Structures of Nonlinear RLC Circuit

	Plain NN	*Block structured NN*	
Net structure:	Single hidden layer	Double hidden layer	
		Layer 1:	*Layer 2:*
Activation functions:	Hyperbolic tangent	Linear	Hyperbolic tangent
Input variables:	$i_L(t-1), v_C(t-1), U(t-1)$	$i_L(t-1), v_C(t-1), U(t-1)$	Not assigned
No. of hidden nodes:	10	2	10
Degree of freedom:	62	60	
Output variables:	$i_L(t), v_C(t)$	$i_L(t), v_C(t)$	

In order to generate the common training data set of the NN models, a simulation was run with the original RLC model implemented in Matlab, with the following simulation settings:

- Simulation time step, Δt: 1×10^{-5} sec.
- Simulation end time: 0.2 sec.

- Simulation input U: Series of 6 different step variations whose values are uniformly picked from $U \in [0,\ 10]$ (in voltage). Those uniformly picked values are permutated before use.

The values and variation of U during the simulation is shown in Figure 39(c). During the simulation, the system state variables i_L and v_C were recorded as the output data. For the NN models, the order of delays for feedback of the outputs is just one, so preprocessing of the simulation result data that is described in §3.3.2.1 was not necessary.

In addition to the simulation for generating the training set of the NN models, another simulation was performed to generate the test data set. In this simulation, the source voltage U was just randomly changed for six times from $U \in [0,\ 10]$.

The training processes of the NN models were performed with the training data set obtained. For each surrogate model configuration, the training was performed five times with this common training data set, in order to evaluate the effect of the model structures to the performance and stability of the generated models. The training was set to stop if either the training MSE reached 5×10^{-9}, or the training iteration achieved the maximum epochs, which was set as 500. As the training algorithm, LM algorithm was chosen for all the NN models.

Tables 9(a) and 9(b) are the training results of the plain and the block structured NN models respectively. According to the training results in Table 9, the block structured NN models outperform the plain NN models based on both the average and the best training MSE of the five trained NN models for each structure. This indicates that the modified block structure actually helped achieving better model accuracy that the plain SHL structure. Figure 38 shows the plots of actual vs. predicted outputs of the two NN models with the best test MSE, which are picked from the two groups. Although the training results in Table 9 shows the block structured NN model has better model accuracy, the plain NN model also seemed to provide good

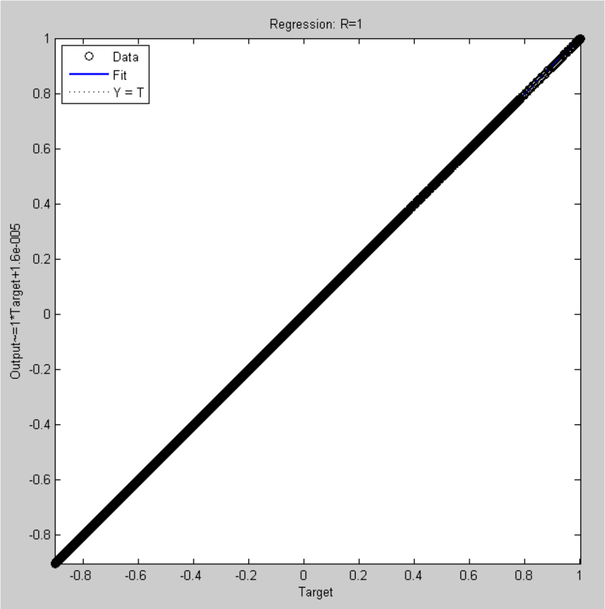
Table 9: Training Results from Two Groups of NN Models of Nonlinear RLC Circuit Model

(a) Plain NN

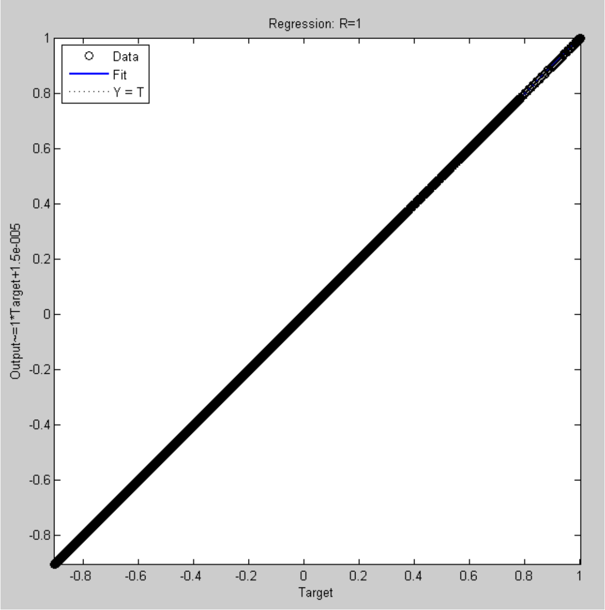
Trial No.	1	2	3	4	5	Average	Best
Training set MSE:	6.2377×10^{-9}	7.4408×10^{-9}	1.2542×10^{-8}	7.4053×10^{-9}	6.3166×10^{-9}	4.7989×10^{-9}	6.2377×10^{-9} (No. 1)
Test set MSE:	4.5546×10^{-8}	1.8254×10^{-8}	2.5471×10^{-8}	4.6895×10^{-8}	1.4477×10^{-8}	3.0129×10^{-8}	1.4477×10^{-8} (No. 5)
Training time (sec):	322.1	318.8	324.6	320.1	318.0	320.7	318.0
Epochs:	500	500	500	500	500	500	500
Training stopped by:	Max. epochs	Max. epochs	Max. epochs	Max. epochs	Max. epochs		

(b) Block Structured NN

Trial No.	1	2	3	4	5	Average	Best
Training set MSE:	4.9870×10^{-9}	4.9853×10^{-9}	1.0839×10^{-8}	4.8272×10^{-9}	4.9889×10^{-9}	6.1255×10^{-9}	4.8272×10^{-9} (No. 4)
Test set MSE:	1.2892×10^{-8}	9.8326×10^{-9}	8.9613×10^{-9}	6.1393×10^{-9}	1.2024×10^{-8}	9.9698×10^{-9}	6.1393×10^{-9} (No. 4)
Training time (sec):	26.4	59.9	183.5	132.6	73.5	95.2	26.4
Epochs:	72	166	500	365	198	260.2	72
Training stopped by:	Max. epochs	Max. epochs	Error criterion	Error criterion	Error criterion		



(a) Plain NN, Trial No. 5



(b) Block Structured NN, Trial No. 4

Figure 38: "Actual Vs. Predicted" Plot of Two NN Models of Nonlinear RLC Circuit with Test Data Set

enough accuracy, at least, based on the static evaluation of Figure 38.

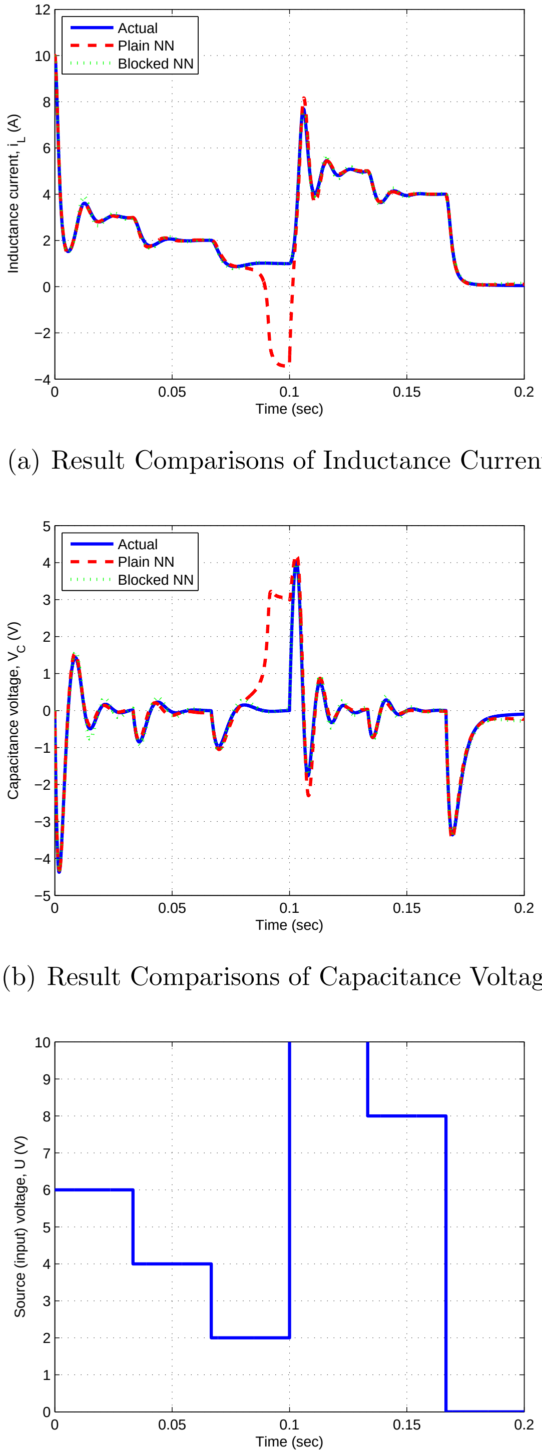
The training speed of the block structured NN models was also significantly faster than that of the plain NN models. In Figure 9(b), the average training time of the block structured NN models was 95.2 seconds, which is about 3.4 times faster than that of the plain NN models. The best training time was about 12 times faster too. These numbers indicate that the modified structure improved both the model training performance and efficiency significantly.

3.4.1.2 Simulation Test of NN Models

Since the static evaluation of the training error of an NN model does not show the model stability and robustness in simulation, The NN models of the two different structures were also tested through the simulation runs. Firstly, the NN models with the best training MSE were picked from the model groups of the two structures, and ran with the input U variations that were used for generating the training data set. The variations of U during the simulation test is shown in Figure 39(c). Figure 39 is the comparison of the simulation results from the plain NN model, the block structured NN model, and the original Matlab model.

Based on the training performance results in Table 9(a) and Figure 38(a), the plain NN model with the best training MSE seemed to have good enough model accuracy for reliable simulation, but the simulation result in Figure 39 showed it was actually not. At about 0.07 second time point of the simulation, the plain NN model began to yield the simulation error rapidly, and then this large error disappeared after the simulation passed about 0.1 second time point. On the other hand, the block structured NN model provided an accurate simulation result that was closed to the result of the original Matlab model.

In order to test whether the problem observed from the plain NN model is a general symptom of the plain NN models in this RLC circuit example, and the greater model stability of the block-structured NN model is true to all the other block-structured models, all the five NN models of each NN structure were ran for the simulations. This time, the variations of input U in the generation of the test data set were applied in the simulations, so both input U and the responses of the original model



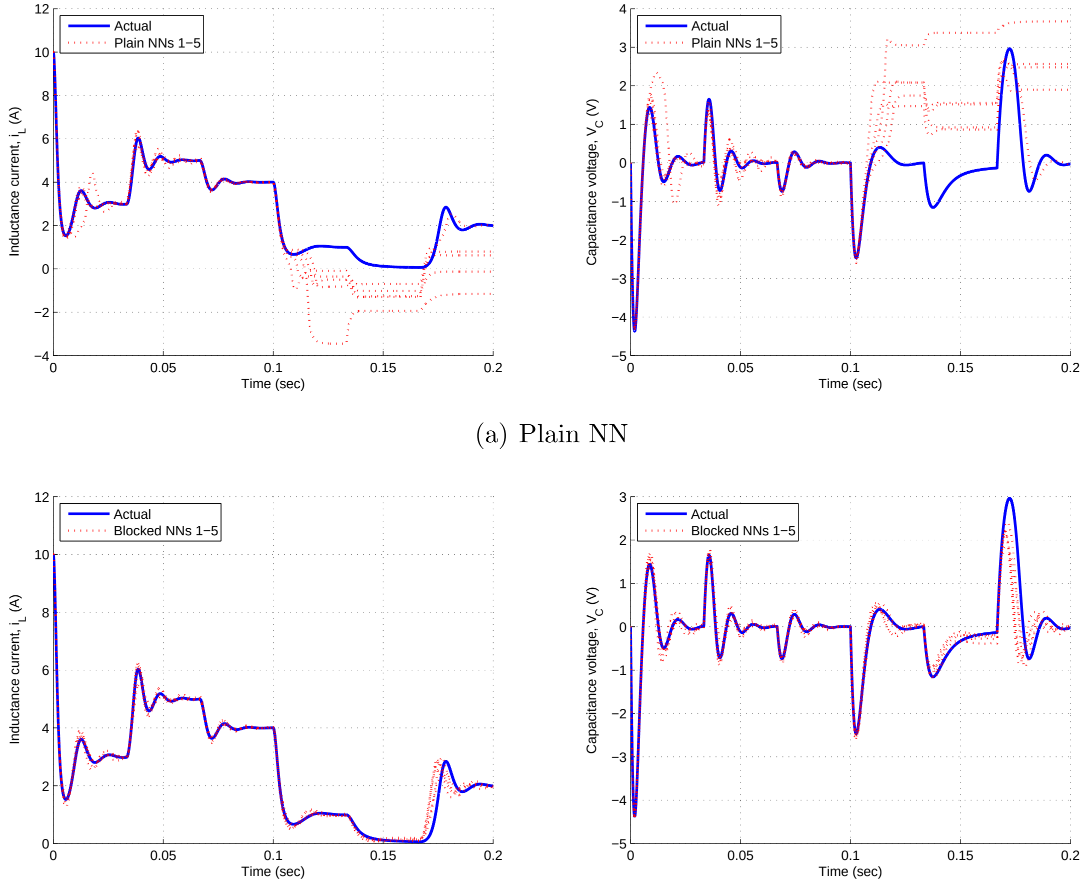
(a) Result Comparisons of Inductance Current

(b) Result Comparisons of Capacitance Voltage

(c) Source (Input) Voltage Variation During Simulation

Figure 39: Training Results of Two NN Models with the Best Training MSE for Nonlinear RLC Circuit

can contains the patterns that were not included in the training data set. Figure 40 is the simulation results.



(a) Plain NN

(b) Block Structured NN

Figure 40: Simulation Runs of the Two NN Model Groups of Nonlinear RLC Circuit with Test Set

As shown in Figure 40(a), all five plain NN models apparently lost their model stability and produced severe errors at the same simulation time point, which is about 0.11 seconds. Interestingly, the inductance current i_L was within the range of 0 to 2 when the models lost their stability in the two simulation tests in Figure 39 and Figure 40. Therefore, it is speculated that the cause of the model instability may be related to a certain region of the output-feedback values, but identifying the exact cause will need a furthermore analysis for investigation. On the contrary, all the block structured NN models yielded the simulation results with acceptable accuracy. This simulation results safely lead to the conclusion that the developed block structure can provide better training performance and efficiency, model stability, and higher robustness for output prediction in surrogate modeling for the RLC circuit example.

3.4.2 Heat Exchanger Unit with Flow Control Valve

The second example is one of the heat exchanger units of CW-RSAD, which is shown in Figure 41. The model has two input variables, which are the valve opening ratio O_v and the pressure difference ΔP held though the two ends of the heat exchanger pipeline. O_v has a scale of 0 to 1, where 0 means that the valve is completely closed, and 1 means that it is completely open. The volumetric flow rate q of the model is the system state variable, which is chosen as the output of the surrogate models in this example. Table 10 is the specification of the heat exchanger unit.

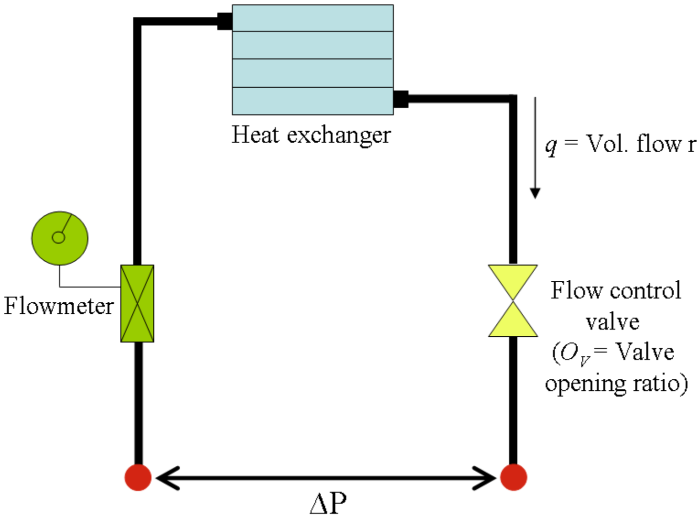


Figure 41: Diagram of Heat Exchanger Unit Example

Table 10: Heat Exchanger Unit Specification

Pipeline	*Heat exchanger*	*Valve*
Tot. pipe length: 12 ft Diameter: 0.5 in	Pipe area: 0.197 in^2 Loss coeff.:1.5 Hydraulic Diameter:0.5 in	Diameter: 0.5 in

The original model of the heat exchanger unit was created using Flowmaster®V7, and this Flowmaster model was connected to Matlab using a COM interface, in order to perform the computer experiments and process the simulation data for generating both training and test data sets. Then, as done in the example of the nonlinear RLC circuit, the NN models were generated using those data sets in the Matlab environment.

3.4.2.1 Generation of NN Models

The NN models were configured in the similar way to the first example case in §3.4.1.1, and the NN model configuration specification is given in Table 11. The differences are mainly on the connection of the input variables to the block structured NN model. In Flowmaster V7, the effect of valve opening ratio O_v to volumetric flow rate q is modeled statically, therefore, their relationship is static in the training data too. This means that, based on the approach in §3.2.4, O_v becomes the static variable, while ΔP remains as the dynamic variable. Thus, according to Figure 32 and §3.2.4, O_v was given as the input to the nonlinear second hidden layer, while ΔP was assigned as the input to the linear first hidden layer of the block structured NN model.

Table 11: Configuration of the Two NN Surrogate Model Structures of Heat Exchanger Unit

	Plain NN	*Block structured NN*	
Net structure:	Single hidden layer	Double hidden layer	
		Layer 1:	*Layer 2:*
Activation functions:	Hyperbolic tangent	Linear	Hyperbolic tangent
Input variables:	$q(t-1), \Delta P(t-1), O_v(t)$	$q(t-1), \Delta P(t-1)$	$O_v(t)$
No. of hidden nodes:	10	1	10
Degree of freedom:	51	45	
Output variables:	$q(t)$	$q(t)$	

Because there were both static and dynamic variables in the regression vector of the block-structured NN surrogate models, the computer experiment was performed with the two-stage experimental design introduced in §3.3.1, in order to generate the training data set. However, there is only one factor for each stage, both the static and dynamic DOE were created just by uniformly distributing sample points, instead of using the modified factorial DOE in Figure 35. For the first stage DOE with the static variable, 11 sample points of O_v were uniformly placed in the range of [0, 1], and for the second stage with the dynamic variable, 13 sample points of ΔP were uniformly picked from $[-5 \times 10^{-4},\ 1 \times 10^{5}]$. Consequently, the generation of the training data set was performed with 11 simulation runs with different O_v values, in each of which the sample points of ΔP were randomly permutated, and used as the scenario of the ΔP variations during each simulation. For the generation of the test data set, a single simulation was run with the 10 random sets of the O_v and ΔP pair as the input scenario.

The training processes of the two groups of the NN models were performed with the training data set obtained. As was in the first example case in §3.4.1.1 the training was performed five times for each surrogate model structure with the same training data, in order to evaluate the effect of the model structures to the performance and stability of the generated models. The training was set to stop if either the training MSE reached 5×10^{-9} or the training iteration achieved the maximum epochs, which was set as 500. As the training algorithm, LM algorithm was chosen for both NN models. The training results were given in Table 12.

Tables 12(a) and 12(b) are the training results of the plain and the block structured NN models respectively, and these training results are consistent with those of the RLC circuit example in §3.4.1.1, showing that the block structured NN models outperformed the plain NN models in both the average and the best training MSE. Figure 38 shows the plots of actual vs. predicted outputs of the NN models with the best test MSE, which were picked from the two NN model groups with different model structures. Although not significant, Figure 38 shows that the outliers from the test of the block structured NN model are less spread than those from the plain NN

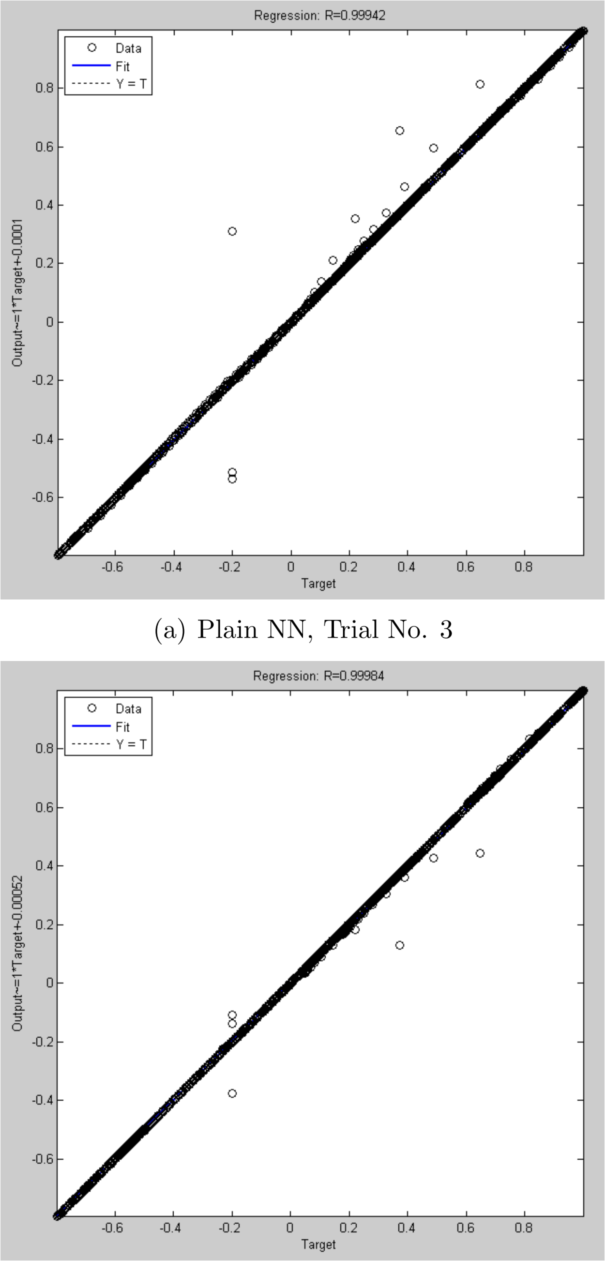
Table 12: Training Results of Two NN Model Groups of Heat Exchanger System

(a) Plain NN

Trial No.	1	2	3	4	5	Average	Best
Training set MSE:	3.5609×10^{-6}	8.8035×10^{-6}	4.6208×10^{-6}	3.4656×10^{-6}	3.2052×10^{-6}	4.7312×10^{-6}	3.2052×10^{-6} (No. 5)
Test set MSE:	3.0385×10^{-4}	3.1160×10^{-4}	2.6629×10^{-4}	2.9953×10^{-4}	2.8758×10^{-4}	2.9377×10^{-4}	2.6629×10^{-4} (No. 3)
Training time (sec):	104.8	104.4	104.4	104.6	104.4	104.5	104.4
Epochs:	500	500	500	500	500	500	500
Training stopped by:	Max. epochs	Max. epochs	Max. epochs	Max. epochs	Max. epochs		

(b) Block Structured NN

Trial No.	1	2	3	4	5	Average	Best
Training set MSE:	2.7326×10^{-6}	1.7462×10^{-6}	9.9629×10^{-7}	9.9897×10^{-7}	9.9347×10^{-7}	1.4935×10^{-6}	9.9347×10^{-7} (No. 5)
Test set MSE:	2.1417×10^{-4}	7.2219×10^{-5}	9.4172×10^{-4}	9.0177×10^{-5}	1.1590×10^{-4}	2.8704×10^{-4}	7.2219×10^{-5} (No. 2)
Training time (sec):	37.2	37.3	15.9	28.1	6.2	24.9	6.2
Epochs:	500	500	212	371	80	332.6	80
Training stopped by:	Max. epochs	Max. epochs	Error criterion	Error criterion	Error criterion		



(a) Plain NN, Trial No. 3

(b) Block Structured NN, Trial No. 2

Figure 42: "Actual Vs. Predicted" Plot of Two NN Models of Heat Exchanger System with Test Data Set

model, reflecting the block-block structured NN model has the better model mapping accuracy, but the plain NN model also seemed to provide good enough accuracy.

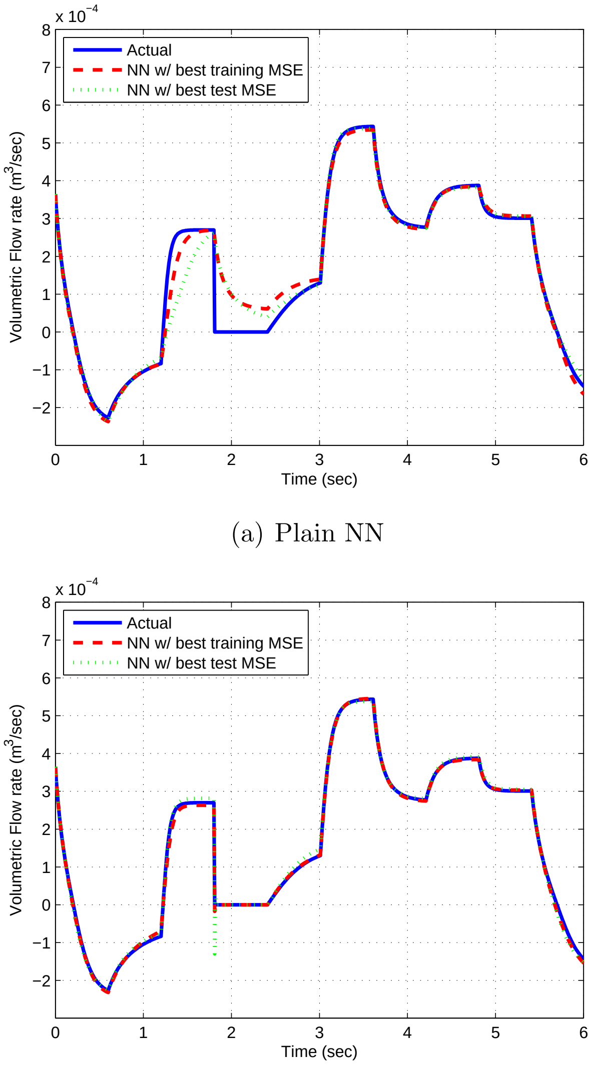
The improvement of the training speed from the block structured NN model was also similar to that of the result from the first example test. From Table 12, the block structured NN models' average training speed was 24.9 seconds, which was about 4.3 times faster the plain NN models, and their best was 9.2 seconds, which was 11.3 times faster than the plain NN models.

3.4.2.2 Simulation Test of NN Models

For this heat-exchanger modeling example, the simulation test based on the training data set was not performed because the size of the training data set was too large to be shown here. Instead, the simulation test was only done using the test data set. Figure 43 is the simulation result from two plain NN models, two block structured NN models, and the original Matlab model of the heat exchanger system. For the NN models, only those with the best training and test MSE were tested from the ten NN models created.

In this second test, both the plain and the block structured NN models did not have large model errors, which were seen from the plain NN models of the first example case in Figure 40(a). It was probably because the heat exchanger system had less demanding dynamics, which was relatively slow and monotonous, than the RLC circuit's fast and oscillatory dynamics.

However, from about 1.2 to 3 seconds in the simulations, there were relatively large errors from the result of the two plain NN models, which were not found from the simulations of the block structured NN models. Interestingly, during that time period, O_v had a very small value, which was 0.1298 for $t = 1.2$ to 1.8 seconds, and 0.0 for $t = 1.8$ to 2.4 seconds. This implies that the relative poor model accuracy occurs when O_v value is very small or zero. The reason may be from the plain NN's experiencing difficulties in mapping the static but very strong nonlinear relationship between the valve opening ratio and the flow. When the valve is being completely closed, the flow friction through the valve is considered to be infinity, and in this condition, the flow value becomes zero no matter how much ΔP is given to the



(a) Plain NN

(b) Block Structured NN

Figure 43: Simulation Runs of NN Models with Another Test Set

system.

In contrast, such large errors do not exist in the simulation results from the block-structured NN models, implying that the output-feedback block structure, and the approach of handling dynamic and static variables separately, improve the accuracy of an NN model significantly. Although the block structured NN models outperformed the baseline plain NN models in the overall model accuracy, Figure 43(b) shows that these block structured models also have some considerable errors. At 1.8 second time point of the simulation, the two block structured NN models have very sharp and large overshoot of the errors. Thus, addressing this problem can be a part of future research. In this research, the following simple correction will be added, when such a problem is encountered in the application of the developed surrogate modeling approach; if $O_v = 0$, q is also set to 0.

3.4.3 Conclusions

In order to test and validate the RNN-based surrogate modeling method for dyanmic system components, the example study was performed with the two simple nonlinear systems, which were the RLC circuit with nonlinear inductance and the heat exchanger unit with a single flow control valve. Setting the SHL Jordan-type RNN as the baseline surrogate model structure, both the baseline and the newly designed block structures were applied to generate the surrogate models of the two systems, for demonstration, and the model training speed, stability during simulation, and accuracy were compared between the models with the baseline and the block structure, in order to validate the developed surrogate modeling method introduced in this chapter.

The study showed that, at least for the cases of the two systems in the example study, applying the developed surrogate modeling method, with the newly designed block-oriented RNN structure, improved training speed, model stability, and model accuracy over the surrogate modeling approach with the baseline surrogate model structure. Despite the successful result of the example study, this result is still not solid enough to conclude that the block-oriented RNN structure can improve the surrogate models' training speed and model stability in all cases of nonlinear system applications, since there is always a possibility of finding counter examples. It is simply a limitation of an experiment-based, inductive validation approach.

Consequently, the study about the model structure-based approach for improving performances of surrogate modeling in this chapter does the role of opening up, or initializing, further future research for maturing such an approach, and solidifying the theoretical background. These future research activities will include either performing a larger number of test cases using the developed method, or deriving the analytical proof for solidifying the validation. Another good research direction is to investigate many other block structures than the Wiener inspired structure. At this point, the surrogate modeling method developed in this chapter will be used as the framework for generating the behavioral models of a naval fluid system, in the final implementation example in Chapter 5.

CHAPTER IV

GRAPH-BASED TOPOLOGICAL AND DAMAGE MODELING

In the developed M&S environment, a graph topological model does the role of weaving all component surrogate models together and enabling a systematic approach of damage modeling and model reconfiguration. In this chapter, the theoretical background and formulation of graph-based topological modeling, and subsequently, the algorithmic tools for the damage analysis of the fluid systems of a military ship.

4.1 Graph-Based Topological Modeling

As introduced in §1.4.4, graph theory has been used for many different fields, and among them, the graph application of linear electrical networks [16, 17] was worth a special attention since it had well-developed and matured modeling methodologies, and electrical networks had a striking analogy with fluid networks so it could provide a starting point of developing the M&S method of a fluid network for damage analysis.

An electrical network can be represented in the form of a graph, like the simple example given in Figure 44. In the graph of Figure 44, its edges are denoted by

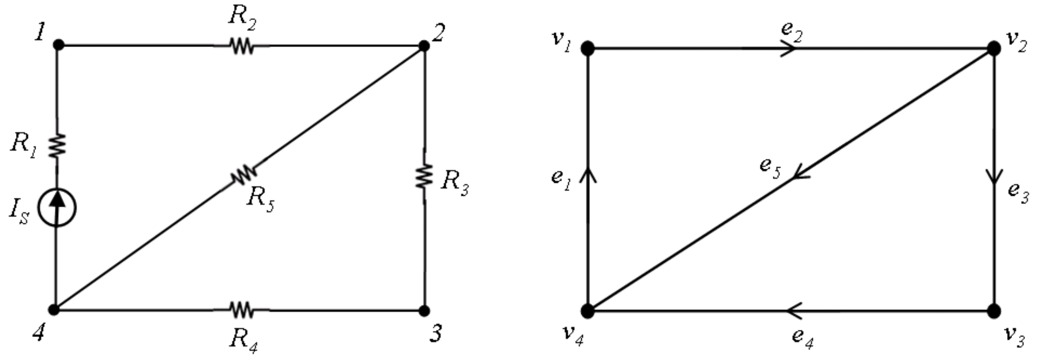


Figure 44: Simple Electric Network and Its Digraph

e_i and nodes by v_j, where $i = 1, \ldots, 5$ and $j = 1, \ldots, 4$. Comparing the graph with its original electrical network model, it can be found that edges represent the actual physical components, and nodes are the entities for defining common boundary conditions between the adjacent components.

The graph in Figure 44 is especially referred to as a *digraph*, a graph with nominal directivity as properties of the edges. Here, the term "nominal" means that the direction is not necessarily that of the actual flow in the system but just a basis of the sign convention upon which the actual flow direction is indicated numerically. For instance, if there is an electric current i_2 on e_2 of the digraph in Figure 44 and its flow direction is opposite of the nominal direction of e_2, i_2 is a minus value.

Graph theory employs the notion of through and across variables in order to describe the state of a system, but in some application domain, it is more proper to use the notion of flow and potential variables instead of through and across variables. Flow variables are basically identical to through variables. Potentials can be expressed in two different forms, one is the node potentials which are the same as potential properties in typical physical systems, and another is the edge potentials that are simply the differences between the node potentials at the two adjacent nodes of an edge. Based on the above definition, it can be known that edge potentials are the same as across variables.

For an electrical network, flows are the currents, node potentials the node voltages, and edge potentials the voltages (i.e., differences of the node voltages on edges) in the network. Similarly, for a fluid network, flows are the flow rates, node potentials the pressures at nodes, and edge potentials the pressure differences on edges. Figure 45 shows the flow and potential variables in the example of a electric resistor component.

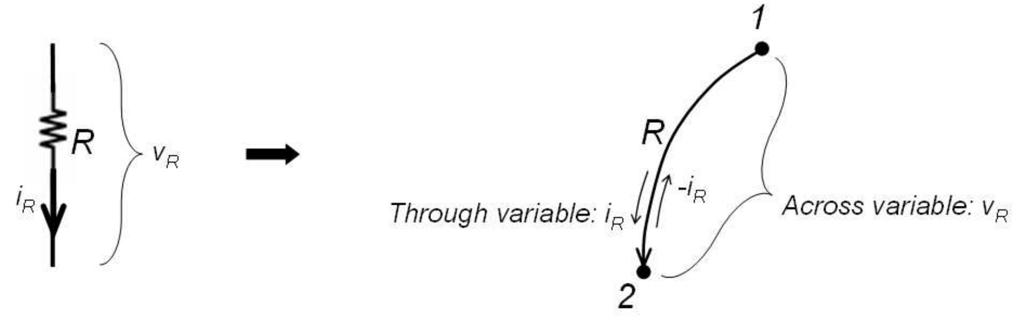


Figure 45: Digraph with Flow and Edge Potential Variables for a Resistor

4.1.1 Basic Mathematical Denotations of Graph

Here, a few mathematical definitions of a graph that will be useful in the later sections is briefly introduced. In this thesis, the mathematical definitions, symbols, and denotations of graph theory are based on Diestel [18], Bollobás [14], and Deo [17].

A graph G is defined as an ordered pair of disjoint sets (V,E) such that $E \subseteq V^{(2)}$, which means, with the superscript of set V, that any element in E is a two-element subset of V). V and E are called the vertex set and the edge set of G, and sometimes V and E are also expressed by $V(G)$ and $E(G)$ to clearly show V and E are the subsets of graph G. The order of G, denoted by $|G|$, is the number of vertices in G. As the more general notion, the number of elements in a set X, called *cardinality*, is denoted by $|X|$. The size of G, which is denoted by $||G||$, is the number of edges in G. In other words, the order of G equals to the cardinality of $V(G)$, which is expressed as $|G| = |V(G)|$, and the size of G equals to the cardinality of $E(G)$, which can be expressed as $||G|| = |E(G)|$.

If $x, y \in V$ are joined by a common edge xy, then they are *adjacent* vertices of G, and *incident* with edge xy. The total number of adjacent vertices of $x \in V$ is the degree of vertex x, denoted by $d_G(x)$ or $d(x)$. The minimal degree of the vertices in graph G is denoted by $\delta(G)$ and the maximal degree by $\Delta(G)$.

4.1.1.1 Incidence Matrix

For a digraph G with $n = |V(G)|$ and $m = |E(G)|$, its incidence matrix, denoted by A, is defined by

$$a_{ij} = \begin{cases} -1 & \text{if } e_j \text{ directs into } v_i \\ 0 & \text{if } e_j \text{ is not incident with } v_i \\ 1 & \text{if } e_j \text{ directs out of } v_i \end{cases} \tag{67}$$

where, $i = 1,\ldots,n$ and $j = 1,\ldots,m$. Again, the signs of the elements in the incidence matrix are determined based on the nominal direction of edges, not the actual direction of the flow in edges.

The following matrix is the incidence matrix of the graph of the simple electric network shown in Figure 44. Each row of the incidence matrix represents a node with the index that corresponds to the index of the row, and in the similar way, each column represents an edge of the graph.

$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & -1 & -1 \end{pmatrix} \end{array} \tag{68}$$

In the incidence matrix, there is always a linearly dependent row in the row space of A. Since a matrix is required to have a full rank in many numerical applications, a reduced incidence matrix, which is obtained simply by erasing the row representing the node with a fixed potential value in A, is used. Assuming that the potential of v_4 is known and fixed, the reduce incidence matrix A_R is expressed as,

$$A_R = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & -1 & 1 & 0 \end{pmatrix} \end{array} \tag{69}$$

which is now linearly independent.

The reduced incidence matrix A_R is very useful for identifying and generating the constraints imposed by the inter-connection topology of a network system. With incidence matrix A_R of the graph model in Figure 44, Equation (70) delivers a complete set of the algebraic equations of Kirchoff's Current Law (KCL) [16], or more generally speaking, the flow-conservation law [70] of an incompressible flow network.

$$A_R \cdot \vec{q} = 0 \tag{70}$$

In Equation (70), $\vec{q}$ is the edge flow vector whose elements represent the flow values on edges, and the sign convention of the flow values in $\vec{q}$ determined based on the edges' nominal directions as described in §4.1. Thus, by applying Equation (70), the KCL equations of the electric circuit model in Figure 44 are expressed as,

$$A_R \cdot \vec{q} = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{pmatrix} = \begin{array}{rcl} -q_1 + q_2 & = & 0 \\ -q_2 + q_3 + q_5 & = & 0 \\ -q_3 + q_4 & = & 0 \end{array} \tag{71}$$

4.1.1.2 Node-to-Edge Potential Transformation

Figure 46 shows the relation between node and edge potentials in the digraph model in Figure 44. In the figure, x_i is the node potential assigned to node v_i, and Δx_j the edge potential on edge e_j, with $i = 1, \ldots, n$ and $j = 1, \ldots, m$. An edge potential is expressed as,

$$\Delta x_j = x_{in}^j - x_{out}^j \tag{72}$$

where, x_{in}^j is the node potential of the in-node – a node from which the edge flow direction begins – of edge e_j, and x_{out}^j the node potential of the out-node of e_j. For example, the edge potential of e_3 is $\Delta x_3 = x_2 - x_3$, as shown in Figure 46.

The relationship between node and edge potentials of a graph can be represented

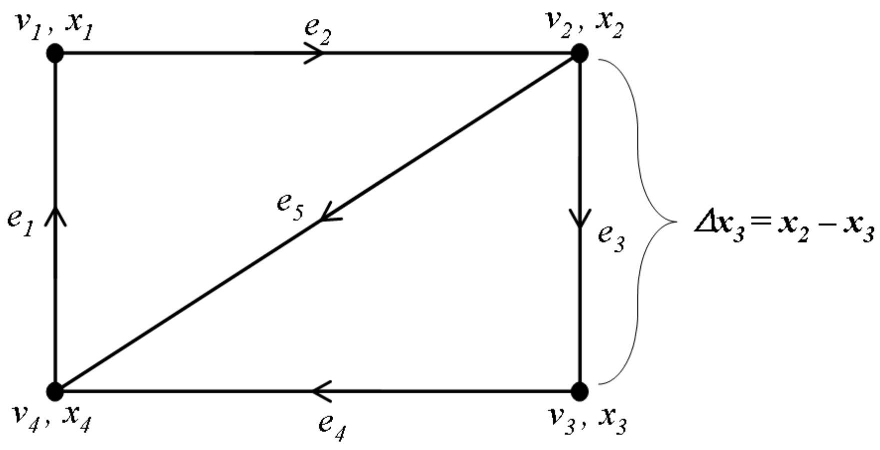


Figure 46: Relations between Node and Edge Potentials

by the following simple equation.

$$\vec{\Delta x} = A^T \vec{x} \tag{73}$$

The matrix transpose of an incidence matrix, $T = A^T$ is also referred to as the transformation matrix. Applying Equation (73), all the edge potentials of the graph model in Figure 46 is,

$$\begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \Delta x_3 \\ \Delta x_4 \\ \Delta x_5 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -x_1 + x_4 \\ x_1 - x_2 \\ x_2 - x_3 \\ x_3 - x_4 \\ x_2 - x_4 \end{pmatrix} \tag{74}$$

4.1.1.3 Laplace Matrix

For a digraph G with $n = |G|$, A Laplace matrix $L(n \times n)$ of G is defined by

$$l_{ij} = \begin{cases} d(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases} \tag{75}$$

where, $d(v_i)$ is the degree of v_i and $i = 1, \dots, n$. The Laplace matrix can also be obtained from an incidence matrix of G by computing

$$L = AA^T \tag{76}$$

Therefore, if the incidence matrix A of a graph G is identified, both the transformation matrix T and the Laplace matrix L of G can be computed with simple linear algebraic operations. Plugging the incidence matrix in Equation (68) to Equation (76), the Laplace matrix of the graph model in Figure 44 is given as,

$$L = \begin{pmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix} \tag{77}$$

As mentioned in §4.1.1.1, the incidence matrix A is lack of the linear independence, so does the Laplace matrix L since $L = AA^T$. Therefore, a reduced Laplace matrix, denoted by L_R in (78), may need in some numerical computing.

$$L_R = \begin{pmatrix} 2 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 \end{pmatrix} \tag{78}$$

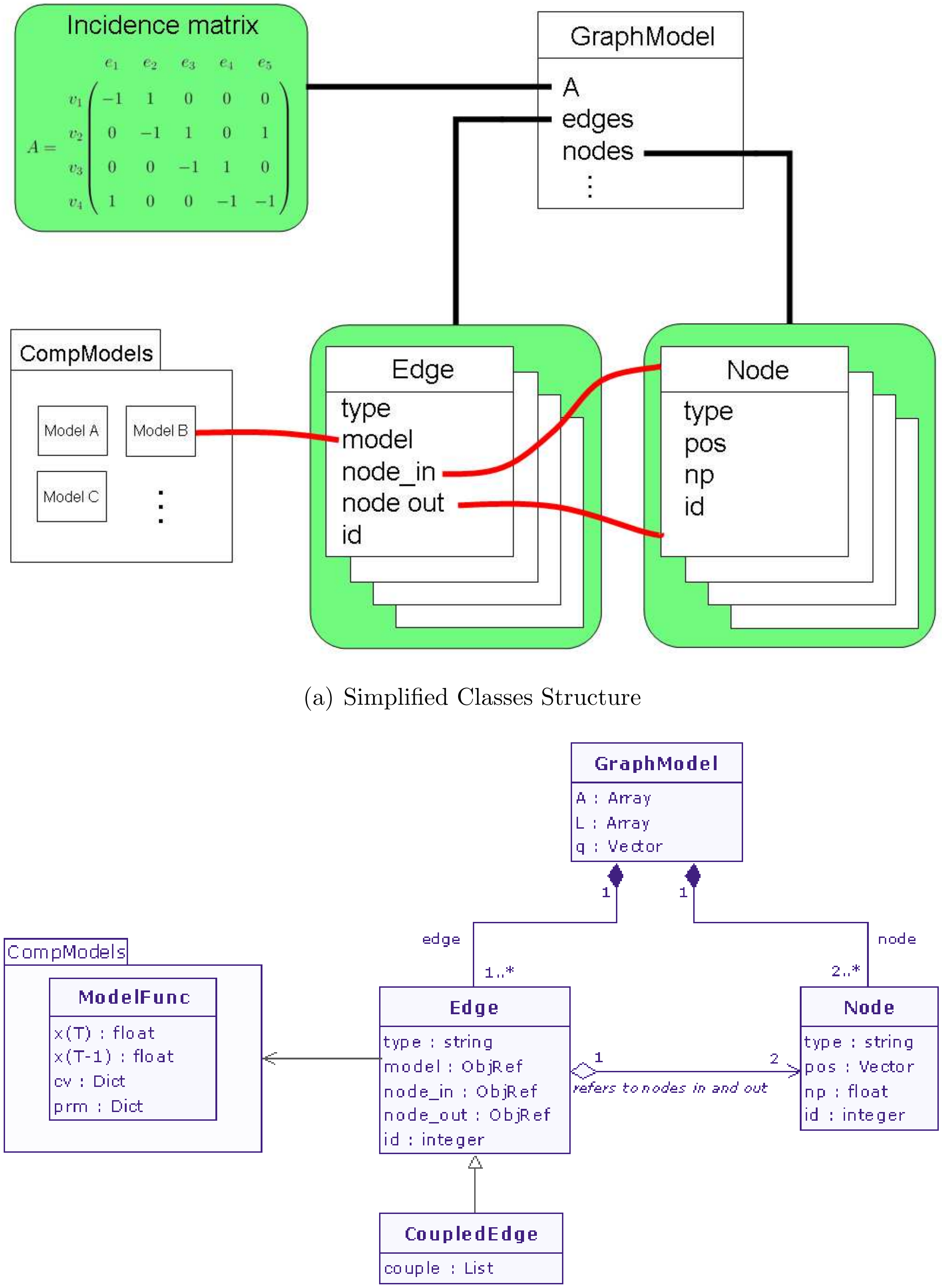
4.1.2 Numerical Implementation of Graph Model

Figure 47 is the basic structure of the object classes in the numerical implementation of graph-based topological modeling. Just for the clarity of describing the class relations, only a few important attributes of the classes were shown in the diagram although there are more attributes and methods defining the classes. The GraphModel class stores the references – variables representing memory addresses – of all edge and node objects using Python's built-in *list* data type. As the essential property representing the connections among all the edges belonging to the graph model, the GraphModel class has its incidence matrix as its attribute. Although not shown in Figure 47, the GrphModel class also has the methods for adding and removing edges and nodes, and generating the incidence matrix by extracting the connectivity information from the edge objects. Whenever the configuration of the graph model is changed by adding or removing edges, the incidence matrix also has to be regenerated to reflect the changes of the connection among them the changes.

4.1.2.1 Edges and Nodes of Graph Model

An edge in a graph always has two nodes incident with itself. If it is an edge of a digraph, then it should also have the directional property. In the implementation of the graph modeling, an edge object has two associated node objects that are linked to it, with the object reference variables in it. The two node objects in the edge object are distinguished as in- and out- nodes, where the in-node is the node from which the edge nominal direction starts, and the out-node is the one to which the edge nominal direction heads.

Flow, node potential, and edge potential variables are also stored and managed in the different objects based on their nature. Node potential values are stored in node objects, and both flow and edge potential values are stored in edge objects. The flow in the edge object needs a physics model which can not be provided by a graph, so each edge object imports a predefined physics model function from the component model library as shown in Figure 47. The *model* attribute in the Edge class in Figure 47 is a “function reference” variable which keeps the memory address of the linked model function in the library and is used for calling the function to compute the flow in the edge. An edge object and a model function are not necessarily one-to-one matching so a single model function can be linked to multiple edge objects, but, of course, an edge object can be associated with a single model function in the library.



(a) Simplified Classes Structure

(b) UML Class Diagram

Figure 47: Elementary Classes for Graph-Based Modeling

Moving the scope more specifically to a fluid system, a static model of incompressible flow in an edge component is generally a function of the pressure difference on the component. Considering that the component may have additional input to the function such as control inputs and model parameters for a design purpose and/or model reusability, the model function has a typical form which can be expressed as,

$$q = f(\Delta P, cv, prm) \tag{79}$$

where f is the model function which is nonlinear, q the volumetric flow rate, ΔP the pressure difference, cv a set of the control variables, and prm a set of the model parameter of the edge component. For a dynamic discrete model case, the model function needs an additional input which is the delayed state variable. Therefore, the dynamic model can be described by

$$\begin{aligned} x(t) &= f\left(x(t-\Delta t), \Delta P(t), cv(t), prm\right) \\ q(t) &= x(t) \end{aligned} \tag{80}$$

where Δt is a model time step, and $x(t)$ is the state variable that is scalar in this case and identical with flow rate $q(t)$. This model function is, in this thesis, created by the surrogate modeling approach which is covered in Chapter 3.

A graph-based model of a fluid system can be developed based on the basic modeling components described above. Let us consider a simple fluid system and the corresponding graph representation shown in Figure 48, where q_i is the flow rate of the ith edge, and P_j the pressure at the jth node.

In the graph-based topological modeling method for fluid systems, edges are distinguished and managed into four types, which are *normal*, *source*, *sink*, and *damaged.* Similarly, every node is in one of three types, which are *normal*, *damage*, and *reference.* The properties of the types of edges and nodes are explained in Table 13. For the fluid model in Figure 48, the translation to the graph representation is straightforward, except for the component with the pump-chiller-reservoir combination, which

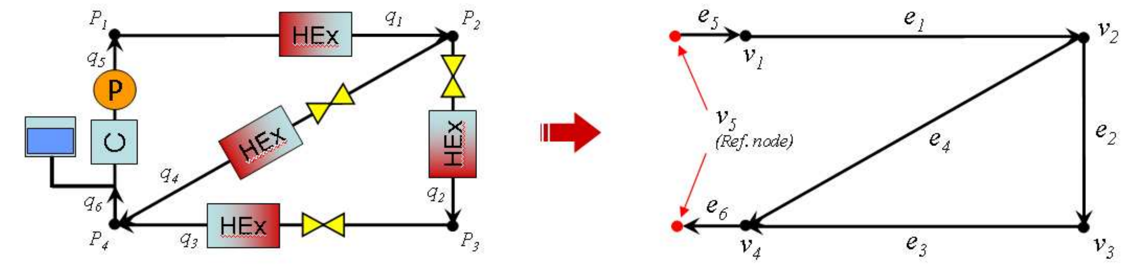


Figure 48: Graph Representation of Simple Fluid Model (HEx: heat exchanger, P: pump, C: chiller)

Table 13: Edge and Node Types

(a) Edge Types

Edge Types	Properties
Normal:	Normal edges.
Source& Sink:	Imaginary edges used to model flow singularities.
Damaged:	Damaged edges. See §4.2 for details

(b) Node Types

Node Types	Properties
Normal:	Normal nodes.
Damage:	Nodes representing the ambiance. Place at the ruptured end of a damaged edge. Their node pressures are set to ambient pressure.
Reference:	Nodes whose pressure values are given and fixed.

could be too complicated to be modeled as a single edge. Therefore, it can be suggested that the component be further decomposed into simpler pieces, and this can be regarded as a reasonable decision for this simple fluid model. The problem is that a realistic fluid system could have significantly more complicated pump-chiller-reservoir networks (see Figure 49 as an example) than the one in Figure 48, having several inner-looping pipelines which make it hard to decompose the subnetwork into a few two-node components. Such an approach also makes the model topology overly complicated for the model's purpose.

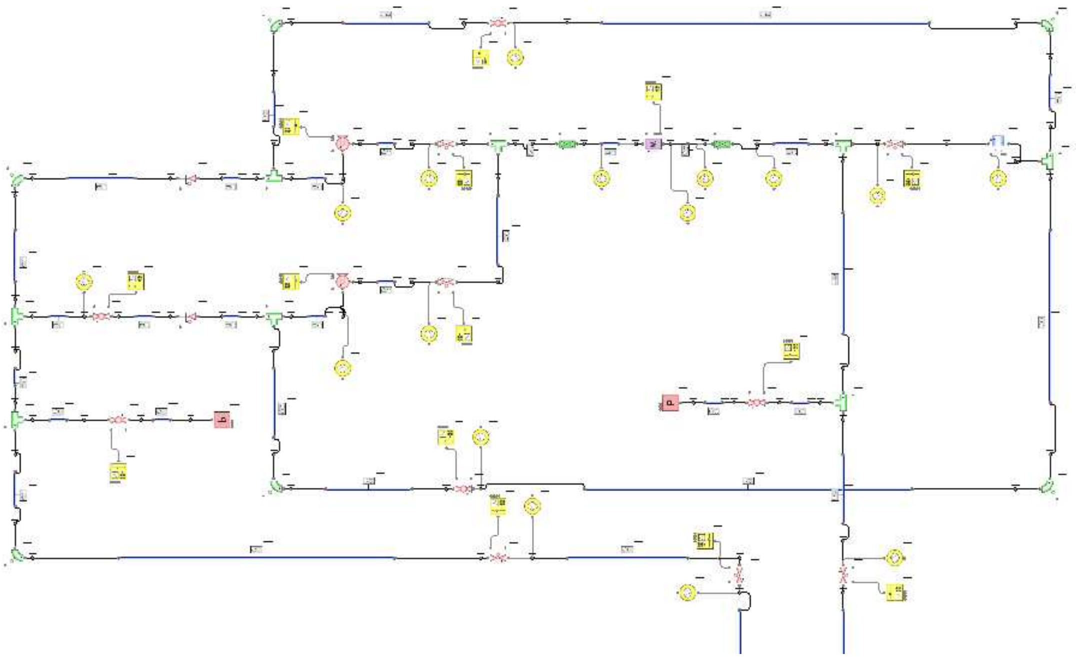


Figure 49: Flowmaster V7 Model of Pump-Chiller-Reservoir Sub-Network in CW-RSAD

When creating a model of the pump-chiller-reservoir unit using a surrogate modeling technique, the component model can not be in the form of Equations (79) and (80) because it is not a function of the pressure difference but the node pressures of the in and out nodes, and the inlet and outlet flow rates are not always the same anymore (e.g., the fluid can shed from the reservoir) in some cases including a system rupture. In order to address this oddity of this component, it can be represented as the combination of two edges – one source type and one sink type – rather than a single edge, as shown in Figure 50. In this setting, the two edges are linked by an artificial reference node with zero-pressure value. By doing so, the pressure differences collected at the two edges are the same as the node pressures of the in and out node of the pump-chiller-reservoir unit, which are, in in Figure 50, $-P_1$ for the sink edge and P_2 for the source edge, and the possible differences between the inlet and outlet flow rates can be successfully expressed by the two edges without violating the unified definition of interfacing between edge components.

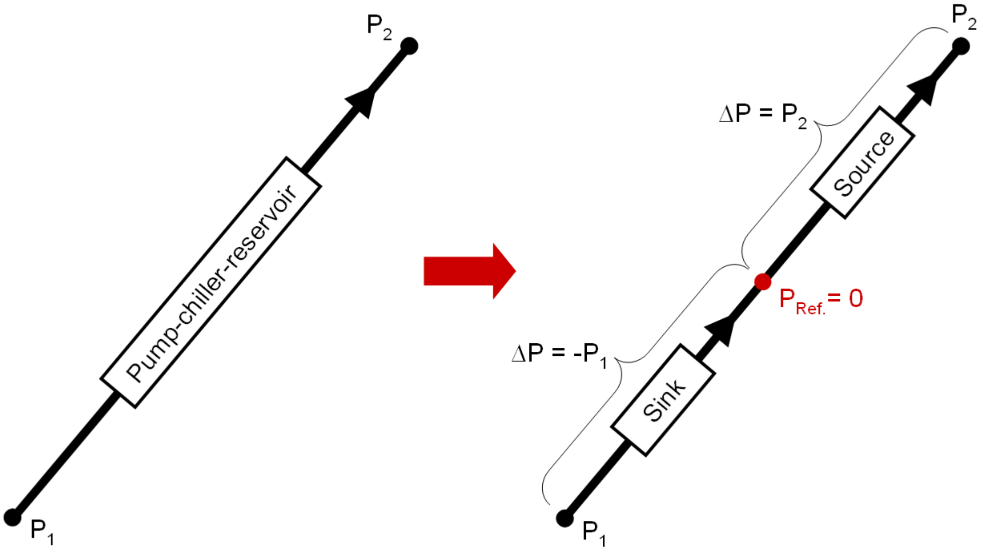


Figure 50: Edge of Pump-Chiller-Reservoir Sub-Network and Conversion to Source-Sink Edge Pair

Remembering that the two edges actually represent the single component, they use the same component model function and are totally coupled in their responses. As a result, they are defined as coupled edge objects using the CoupledEdge class (shown in Figure 47(b)), a derivative of Edge class. A coupled edge stores the list of edges coupled with it in the *couple* attribute and accesses their input and output variables whenever it calls the shared model function for computing the flow rate. Figure 51 describes the objects and their association inside the graph model instantiation of the simple fluid network given in Figure 48.

4.1.2.2 Connectivity Modeling using Incidence Matrix

The GraphModel class has the two incidence matrices A_{tot} and A as shown in Figure 52. A_{tot} is the full incidence matrix of all nodes and edges of a graph model. Assuming the number of normal nodes p, $p \times m$ matrix A in Figure 52 is the reduced version of A_{tot} and contains only the incidences of the normal nodes from A_{tot}.

In order to identify the KCL-based algebraic constraints of the graph model, the GraphModel class uses Equation (70) with A, instead of A_{tot}, because the pressure

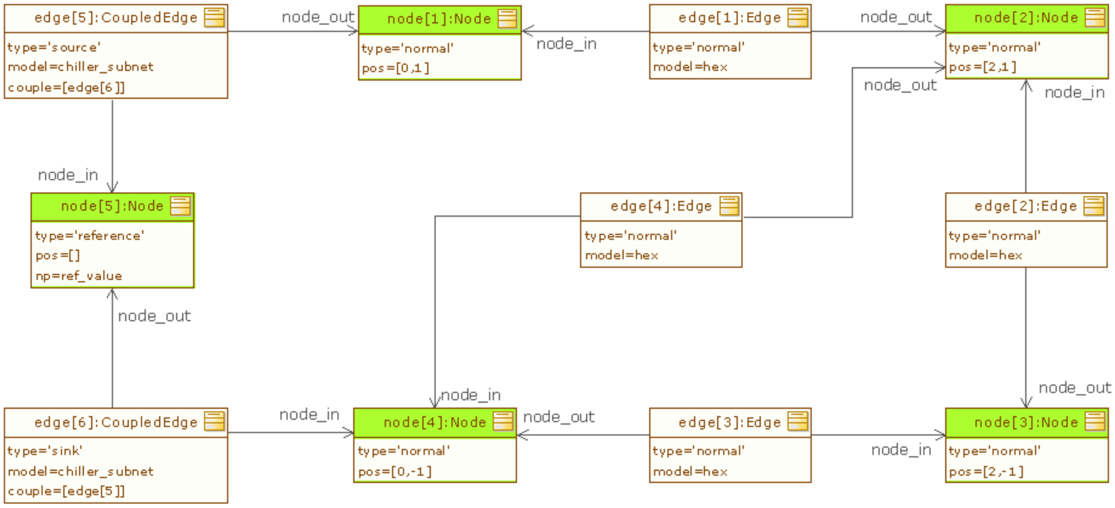


Figure 51: Object Diagram of Simple Fluid Model

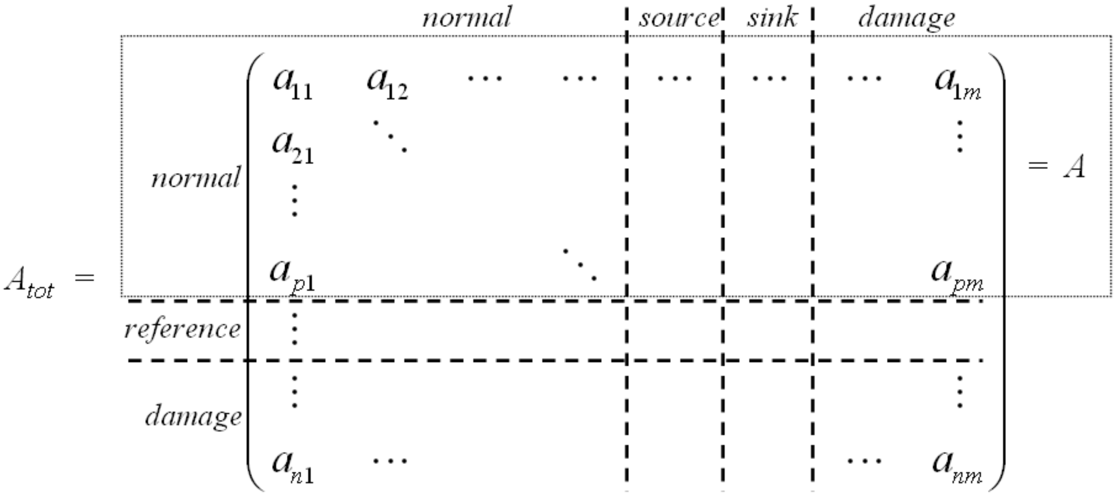


Figure 52: Incidence Matrix Data Structure in GraphModel Class

values of damage nodes and reference nodes are given, and the rows representing damage and reference nodes in A_{tot} are all linearly dependent vectors. However, the other M&S formulations except for this use A_{tot} because A_{tot} contains the complete information of the model connection topology.

4.1.2.3 Topological Layout vs. Geometric Layout

For the design of a physical network such as a fluid and an electrical network, the geometric layout of its components is an important property for the analysis for survivability. However, such a property is also largely unknown in the early design phase or subject to be changed frequently until the detail design phase. In order to handle this vagueness in the knowledge of the geometric layout, the modeling method extensively uses the connection topological layout of the graph model instead of the geometric layout. Here, the topological layout means the relative locations of inter-connected components and connection points.

For the numerical implementation, the *topological coordinates* were created just as the Cartesian coordinates were defined. The difference of the topological coordinates from the Cartesian coordinates is that the points and the distances in the former do not indicate the actual spatial locations and distances but just the relative locations in an imaginary space. For instance, Figure 53 shows all the node objects of a graph model with their topological locations. For edges, the topological locations were set as the middle point of the linear connection of the two incident nodes.

Although the topological layout is lack of the geometric and spatial information of the components and their connections of a system, it can still allow damage analyses of a physical network within a level of fidelity that can be acceptable in an early design phase, since the system response to damages or failures is strongly affected by how the system components are concatenated with the damaged components rather than geometric details regarding the damages/failures and the system components. Consequently, it contributes to bring forward the damage analysis, which was typically possible in the later design processes, to an early design phase.

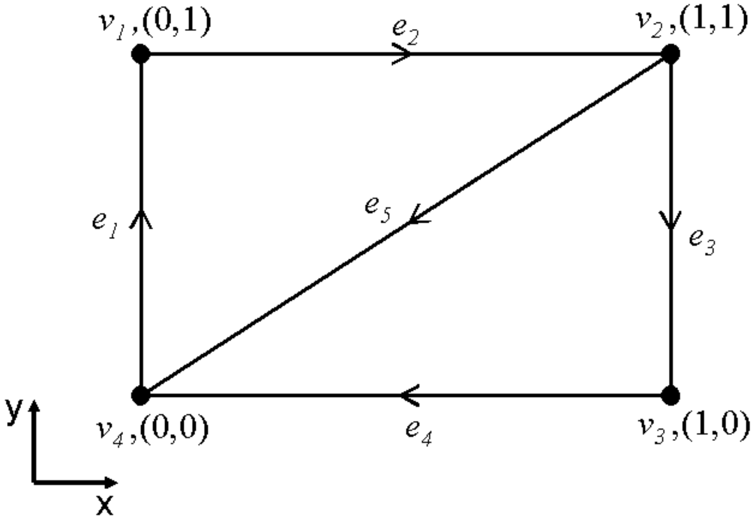


Figure 53: Toplogical Coordinates on Graph Model in Figure 44

4.2 Damage Modeling

The proposed M&S formulation sets two elements for the damage analysis with graph-based modeling of fluid systems. The first one is a damage entity, whose role is to change the topology of a graph model and the properties of the edges and nodes in the graph model, in order to model the system after a damage has occurred. The second element for damage modeling is a damage control model. A damage analysis of a smart actuator-equipped reconfigurable fluid system without any damage control effort is meaningless, since the analysis would yield only a trivial solution of the total system failure by the eventual shedding of all the fluid out. Considering that the control development is mostly done in the later design processes, the M&S environment must include a proper control model as a part of its whole package so it can provide the conceptual or preliminary level damage analysis without having to wait for the control design to be delivered. Lastly, as an essential part of developing the damage control model, an algorithm for auto-generation of damage control model for a given graph model is developed. This auto-generation algorithm is important for keeping the M&S environment highly flexible for automated model reconfiguration in the design-oriented analysis in the early design stages.

4.2.1 Damage Bubble

Since the application is the M&S of a military ship, the causes of damage of interest are explosions from hostile actions, such as projectile shots, anti-ship missile attacks, and mine explosions. Each of These damages on the system can be represented simply by a "damage bubble" object, a sphere with its topological location and volume property. When it is placed in the topological space of the fluid model, the bubble object removes or changes the nodes and the edges that are immersed in it in order to model damage in the system. Figure 54 shows a damage bubble on the fluid model in Figure 48. The process of the damage bubble object for reconfiguring an original fluid system model to a damaged system model consists of the following steps:

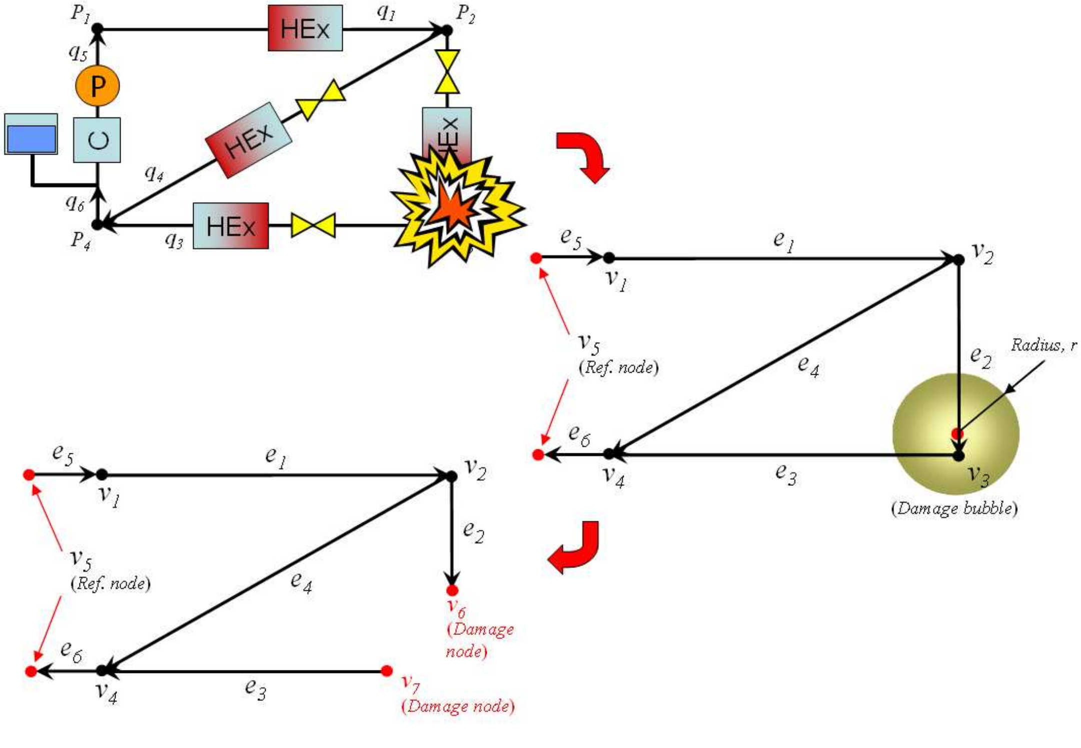


Figure 54: Rupture in Simple Fluid System

1. Create a damage bubble object at simulation time $t = t_{dmg}$. The location, size, and damage triggering time t_{dmg} are all defined by a damage scenario which is

simulation input.

2. Identify the whole or part of each edge body and associated nodes that are inside the damage bubble.
3. Eliminate or modify the identified components of the model.
4. Place damage nodes at the open ends of the damaged edges after their damaged portions were eliminated.

The damage nodes in the last step of the process represent the open-ends of damaged pipelines. When a pipe is ruptured, its ruptured end is exposed to the ambiance whose pressure can be considered to be constant as simplification. In modeling implementation, the damage node objects have a fixed pressure value of 1.1 bar and placed to represent the end of ruptured pipelines open to the ambiance. In the next section, the logics of identifying and applying damages of the component of a graph model is introduced in more detail.

4.2.1.1 Identification and Elimination/Modification of Damaged Components of Graph Model

The logic underlying the identification of the damaged components of a graph model is based on the a simple practice of analytic geometry for finding the intersections between a line and a sphere, which are shown in Figure 55. For simplicity of the M&S, all edges are straight lines in the topological coordinates. For an edge, let the coordinates of the in and out nodes be $X_{in} = [x_{in}, y_{in}, z_{in}]^T$ and $X_{out} = [x_{out}, y_{out}, z_{out}]^T$. Then any point $X = [x, y, z]^T$ on the edge can be expressed by

$$X = X_{in} + t \cdot (X_{out} - X_{in}) \tag{81}$$

When t in Equation (81) is $0 \leq t \leq 1$, point X is on the edge; otherwise, X is on the line extended to outside of the edge, which is drawn as the dotted line in Figure 55.

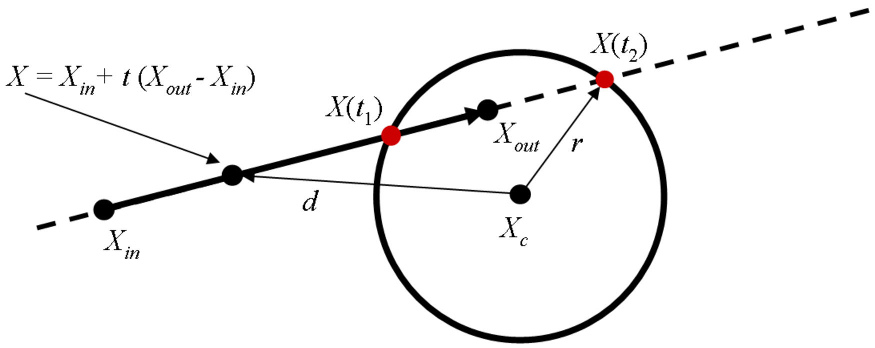


Figure 55: Junction Points between Edge and Sphere

Considering the damage bubble with the center at $X_c = [x_c, y_c, z_c]$ and the radius r, the distance d between a point on the edge and the center of the bubble is,

$$d = \|X - X_c\| \tag{82}$$

or,

$$d^2 = \|X_{in} + t \cdot (X_{out} - X_{in}) - X_c\|^2 \tag{83}$$

By substituting $\Delta X_{ic} = X_{in} - X_c$ and $\Delta X_{oi} = X_{out} - X_{in}$ to Equation (83), and reorganizing it, the distance equation becomes a quadratic polynomial equation of t, which is expressed as,

$$\begin{aligned} d^2 &= at^2 + bt + c \\ where&, \\ a &= \|\Delta X_{oi}\|^2 \\ b &= 2\cdot <\Delta X_{oi}, \Delta X_{ic}> \\ c &= \|\Delta X_{ic}\|^2 \end{aligned} \tag{84}$$

If a straight line intersects with a sphere, there exist one to two intersection points whose distance from the center of the sphere X_c is radius of the sphere r. These intersections can be found by setting $d = r$ in Equation (84) and solving for the roots of the equation $r^2 = at^2 + bt + c$. In the case that there is only one intersection, then

$t_1 = t_2$. Letting t_1 and t_2 denote the two intersection points, the corresponding two intersection points are marked as red points in Figure 55.

However, the existence of intersections does not necessarily mean that an edge, not the line that includes the extension of the edge, is in the sphere. Assuming there exist intersections, the placement of these intersections can be represented by one of the four cases in Figure 56. In Figure 56, L is the length of an edge component, and the intersections are marked by red points. Then, the four intersection placement cases in Figure 56 can be interpreted as the following damage cases:

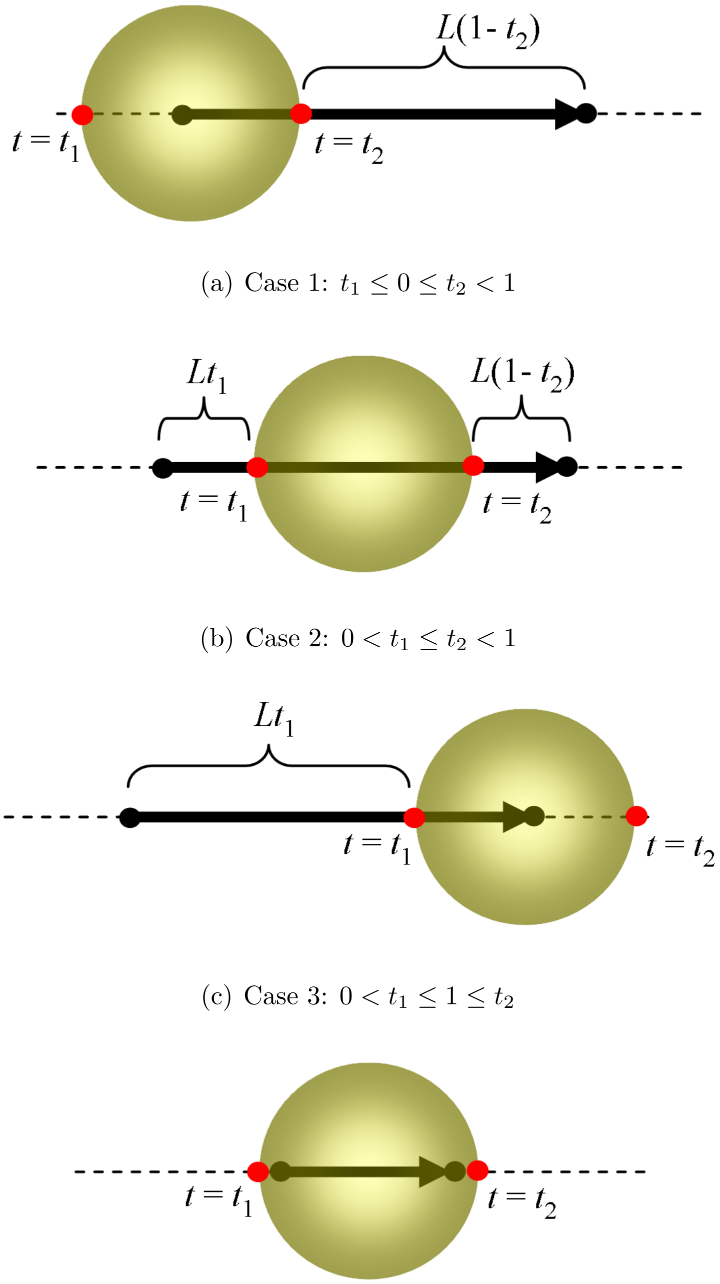
1. $t_1 \leq 0 \leq t_2 < 1$: $L \cdot t_2$ of the edge including the in-node is damaged.

2. $0 < t_1 \leq t_2 < 1$: Only the inner area of the edge is damaged. The edge is now broken into two pieces with the length $L \cdot t_1$ and $L \cdot (1 - t_2)$ respectively.

3. $0 < t_1 \leq 1 \leq t_2$: $L \cdot (1 - t_1)$ of the edge including the out-node is damaged.

4. $t_1 \leq 0$ and $1 < t_2$: The entire edge including the two incident nodes is damaged.

In the M&S, the above logic is implemented as the process of the damage bubble object and used to identify and eliminate/modify the damaged components. The notional damage simulation process with the damage bubble object is described using the UML activity diagram in Figure 57. The actual algorithm is more complicated but follows this notional process. In the activity diagram in Figure 57, yellow square boxes are data entities or objects that are created as a result of activities. These entities are again used as the inputs to other activities. The activity diagram in Figure 57 shows only the process of applying damage using a damage bubble object as simplification. For a description of more comprehensive process flow for damage simulation, see Figure 66. In Figure 57, the dotted box represents the iterative routine of the four case-based damage identification algorithm that was just explained, and this routine is performed for all the edges, applying and updating the changes of



(a) Case 1: $t_1 \leq 0 \leq t_2 < 1$

(b) Case 2: $0 < t_1 \leq t_2 < 1$

(c) Case 3: $0 < t_1 \leq 1 \leq t_2$

(d) Case 4: $t_1 \leq 0$ and $1 < t_2$

Figure 56: Four Cases of Intersection Placement Representing Different Damage Cases of Edge

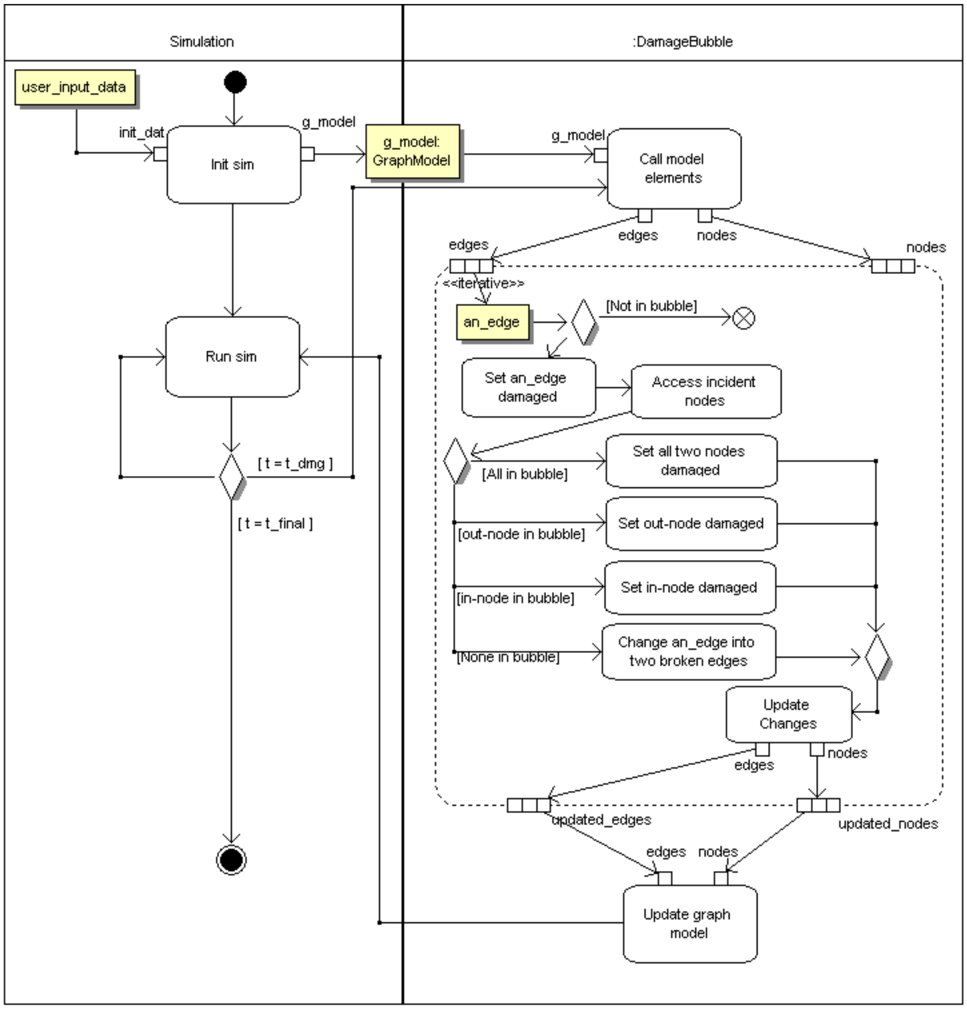


Figure 57: Implementation of Damage Bubble in Simulation

the properties of the edge and the node objects in a graph model object. With this updated graph model object, the simulation is continued to generate the system responses after the damage occurred.

4.2.1.2 Limitations and Possible Expansions of Damage Bubble Application

The damage bubble developed here is a low-fidelity representation of damage by explosions on the body of a naval military system; however, as more information for modeling damage is accessible and higher fidelity damage modeling is necessary, the damage bubble object and its process can be modified to be a more sophisticated representation of such a type of damage. One example case is to include a probabilistic approach in determining damaged elements and the damage levels of them. In the current damage bubble object, the process of identifying damaged components and applying damage in a graph model is performed simply by checking whether the components are inside the damage bubble or not. More realistically, however, the component damages and the intensity of the damage should be represented by probabilistic modeling, rather than such simple “yes or no”-type modeling. Furthermore, the probability of both damage and damage intensity of a component must be a function of various factors, including ship geometry, fire and temperature, time, and component material and reliability. For instance, a component will have higher probability of being damaged as it is closer to the center of a damage bubble. However, if there is a bulkhead between the component and the center of the damage bubble, the probability of being damaged must be decreased.

4.2.2 Reference Damage Control Model

The developed M&S incorporates an element called the *reference damage control model*, which is designed to deliver a near-ideal control performance without the considerations of technological and economic viabilities. The development of such a concept stems from the following two design concerns. First, the development of a reconfigurable engineering system is a multidisciplinary problem involving both the plant design and the control system design. A typical approach to a multidisciplinary problem is the serial or parallel executions of the domain-level iterative design processes in which the designs of the other domain systems are all fixed until they are updated in the system-level iteration. In this approach, a reference control model can be an initial control design when the local iterative process for the plant design is executed. Second, a control baseline is needed in order to measure the performance of the ongoing control design, and the reference control model can be used as a baseline design that provides an estimation of the ideal performance that a control design can achieve.

The reference control model is developed by removing many design constraints so that the model can have the near-maximum performance, but it does not necessarily mean that the model is also free from the key philosophy or paradigm to be achieved from the control system design. Since a highly distributed, agent-based control architecture is the underlying design paradigm used by the Navy, it is kept as the development architecture of the reference damage control model. For the M&S of the ship fluid system, the reference damage control model was developed with the following presumptions that remove the constraints that potentially limit the control performances:

1. There is no communication delay, i.e., no consideration of network latency. This also implies no control instability by signal delays.

2. The communication network causes no noise and bias of signals.

3. The cost of networking is not considered. A controller unit can access any signal required for its process.

4. Perfect sensing. There is no error induced from sensory processing or hardware.

5. There is no control hardware and software induced delay such as micro-controllers' processing time and the inefficiency of control algorithms.

As previously stated, the reference damage control model is a model of a distributed control system. It consists of multiple component-level control units or agents, which are attached at local actuators of the system plant, providing reflexive and collaborative actions in order to isolate the ruptures in a fluid system. The reference damage control model is designed to be single-layered, meaning that it does not have high-level inferences and intelligence which are mostly implemented by multi-layer, hierarchical control architecture. The reasons for applying such a simple architecture are that, 1) the development of a properly working layered control system model alone is a highly difficult and complicated research activity, so it had to be avoided, 2) a simple, single layer control model is good enough for the reactive action of damage isolation and so is for simulation-based damage analysis for plant design, and 3) the layered architecture with higher-level intelligence makes the resulting control system model less reconfigurable and scalable, so the modeling environment becomes less model-reconfigurable.

4.2.2.1 Component-Level Control Agents

The reference damage control model is realized by the aggregation of component-level control units, so developing an entire control system model is highly dependent on proper development of its individual control unit. Every control agent object in the modeling environment basically contains two main information for control action. The first one is the information of the neighboring control units and the operational status of them, and the second one is the internal control process, which is performed based on the status information gathered from its neighborhood. This local control process of each control unit results in collaboration with neighboring controllers and isolation of a rupture. Figure 58 is the simplified description of the control unit object implemented for modeling a controller attached to the component represented by edge e_2.

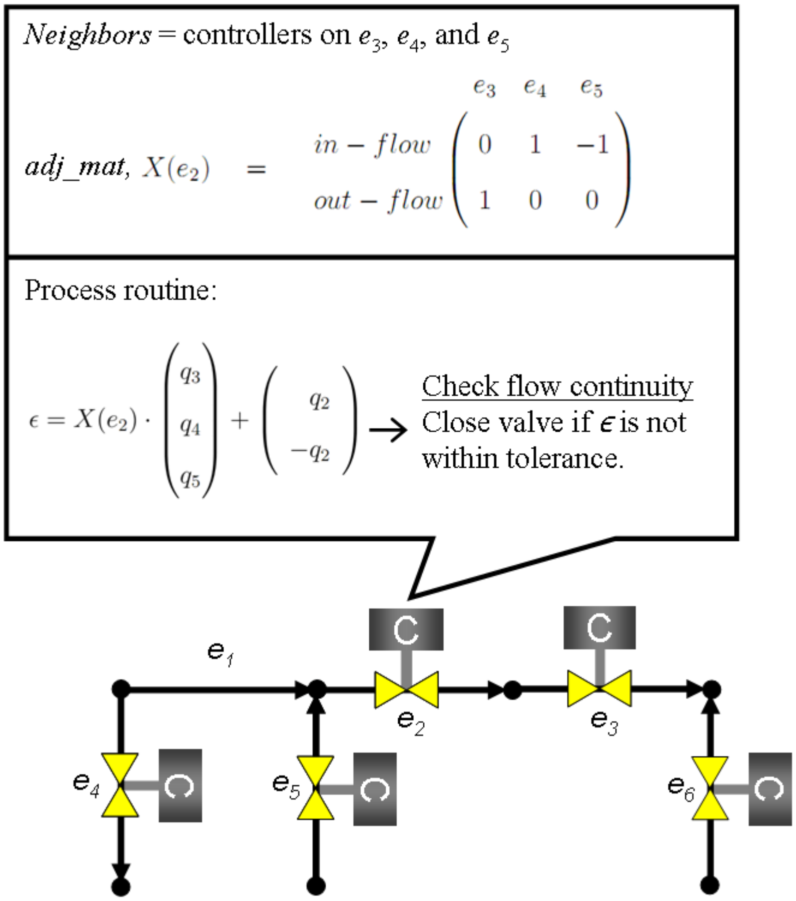

Neighbors = controllers on e_3, e_4, and e_5
$adj\_mat,\ X(e_2) = \begin{array}{c|ccc} & e_3 & e_4 & e_5 \\ in-flow & 0 & 1 & -1 \\ out-flow & 1 & 0 & 0 \end{array}$
Process routine:
$\epsilon = X(e_2) \cdot \begin{pmatrix} q_3 \\ q_4 \\ q_5 \end{pmatrix} + \begin{pmatrix} q_2 \\ -q_2 \end{pmatrix} \rightarrow$
Check flow continuity
Close valve if ϵ is not within tolerance.
C
C
e_1
e_2
e_3
e_4
C
e_5
C
e_6
C

Figure 58: Composition of Component-Level Control Unit of Smart Valve in Simplified View

In the implementation of a control unit object for controlling a valve, the neighbors list attribute represents the physical connections for communication between a controller and its neighbors. The neighbors of a control unit are defined based on their flow-based adjacency to the control unit; for instance, the controller attacted on edge e_2 in Figure 58 has the controllers on e_3, e_4, and e_5 as its neighbors, since they are adjacent with the controller on e_2 based on flow connectivity. Again, the neighbors list represents the physical communication lines or (wireless) connections between the control unti and its neighbors. Through the communication lines, the controller and its neighboring controllers share the measurement of the flow rate at the valves on which they are embedded.

Along with the neighbors list, there is another attribute associated with it. *adj_mat*, which is actually a local version of incidence matrix for the process of an individual controller, contains the information about how a controller is connected with each of the neighbors. *adj_mat* attribute is a two-row matrix. The first row represents the "in-flow" side connections, and the second row represents the "out-flow" side connections with the neighbors, based on the nominal flow direction given to each edge. An example of this local incidence matrix is also given in Figure 58. This matrix is almost identical with an incidence matrix created with only in- and out-nodes of an edge with a controller. The difference is that the nonzero elements in *adj_mat* matrix do not mean the incidences of arbitrary edges but the flow-based connectivities of neighboring controller-attached edges to the two nodes of another controller-attached edge, and the columns of this matrix represents the neighboring controllers, instead of all edges in a graph model.

Using the neighbors list, the flow rates gathered from the neighboring controllers, and the *adj_mat* matrix, the process of a control unit checks whether there is a flow leak by system rupture located between itself and the neighbors. The algorithm is simple; a controller checks the flow continuity between itself and its neighbors by computing a simple matrix operation given as,

$$\epsilon = X(e_i)\vec{q}_{nb}(e_i) + \begin{pmatrix} q_i \\ -q_i \end{pmatrix} \tag{85}$$

where $X(e_i)$ is the local adjacency matrix (*adj_mat*), $\vec{q}_{nb}(e_i)$ is the vector of the flow rates from neighbors, and q_i is the flow rate measured in the controller attached on e_i. If any element of the two dimensional vector ϵ is not close to zero within a numerical threshold ε, the controller assumes that there is flow discontinuity caused by system rupture on the pipeline between itself and its neighbors, and it closes the valve on which it is attached, as a reaction to isolate the rupture. Since every individual controller that identifies a rupture will close its valve, ruptures on the system will be isolated eventually.

4.2.2.2 Numerical Implementation of Reference Damage Control Model

The classes for the reference damage control modeling are shown in Figure 59. DmgCtrlSys class represents a damage control system, and SmartValve and CpAgent classes represent the controller-embedded smart valve units and the control units for chiller-pump sub-networks. As aforementioned, the reference damage control system is basically a non-hierarchically aggregated system of multiple smart valves and controllers of chiller-pump sub-networks. The smart valve object processes its behavior based on the local rules and sensor reading from the surrounding environment such as the flow and damage of the neighboring smart valves, creating reflexive and cooperative actions of isolating ruptures in the system. The chiller-pump network controller is similar to the smart valve except that it controls pump units and multiple valves in the sub-network.

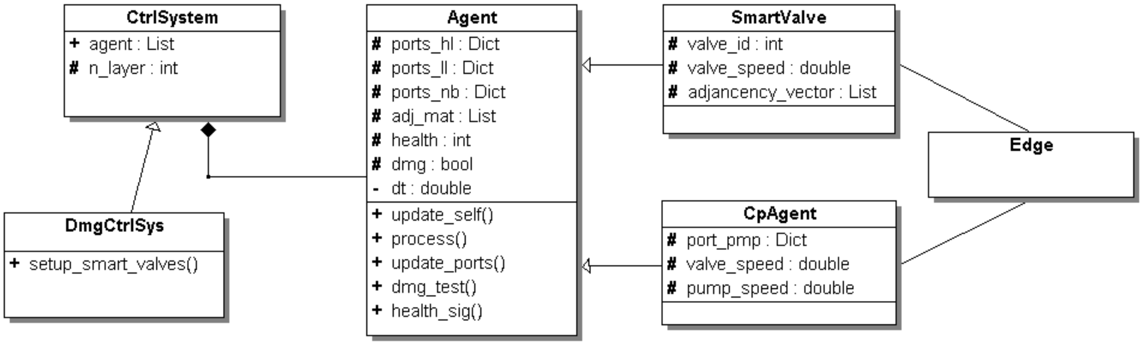


Figure 59: Distributed Reference Damage Control Modeling Classes

In Figure 59, both SmartValve and CpAgent classes have the attributes *ports_hl*, *port_ll*, and *ports_nb* as the communication channels to the control units of their upper layer, lower layer, and neighborhood. For the current application, there is no supervisory hierarchy so upper units do not exist, while the lower-layer units are just the edge components that contain valves or pumps. The neighbors are the other controllers that have direct pipeline connections with it.

The *adj_mat* attribute in SmartValve and CpAgent classes represents the local connectivity to its neighboring controllers. For example, the local adjacency matrix $X(e_2)$ for the valve on edge e_2 in Figure 48 is

$$X(e_2) \quad = \quad \begin{array}{r} \\ in - flow \\ out - flow \end{array} \begin{array}{c} \begin{array}{ccc} e_3 & e_4 & e_5 \end{array} \\ \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & 0 \end{pmatrix} \end{array} \tag{86}$$

where the smart valves on e_3 and e_4 and the chiller-pump net controller attached on e_5 (and e_6) should be in the *neighbor* list of the valve object. Equation (86) is the *adj_mat* matrix of the controller attached to e_2. As previously stated, the two rows describe controller connectivities at the upstream (in-flow) and downstream (out-flow) directions based on the edge direction on e_2, and the column length is the same as the number of neighbors. The elements of the matrix follow the notation of the incidence matrix.

The attribute *health* in SmartValve and CpAgent classes indicates the health condition of a device. Currently, it can have only two different values, which are 1 for the "healthy" condition and 0 for the "destroyed" condition. Once a smart valve object receives a 0 signal for the health condition from any of its neighbors, it closes its valve immediately without checking the local continuity.

4.2.2.3 Limitations and Possible Expansions of Component-Level Control Agent Implementation

The reference damage control model is never created for the intention to model a control system with a high level of reality and sophistication. It is just a numerical tool developed for marginally enabling the damage analysis of a ship fluid plant in the early design stage. Therefore, there are many parts that are abandoned in the development of the control agent model, in order to keep the M&S problem simple and flexible and maintain the modeling efficiency. One of many missing parts in control-agent modeling is the controller's process for managing device failures in order to achieve the better robustness of the ship system. The malfunction of a smart actuator can be caused not only by the external factors such as explosion and physical rupture, but also by the device failure, which is from the component's nature of reliability. For example, the valve actuator of a smart-valve component can be broken and stuck in a certain valve opening states; or the embedded micro-controller in the smart valve can experience physical or software-caused faults, such as short/open-circuit and code bugs. Each control unit and the distributed control system must be able to manage such malfunctions in a way to minimalize the disturbance to the ship system-level operation and performance. When it comes to including the analysis of system robustness to device failures early in the design processes, the reliability data of various devices of interest and probabilistic failure modeling of the plant are also required in that stage. Also, the control laws of identifying device failures and reacting to prevent the ship system from cascading failure have to be developed and added to the local processes of the component-level control units.

4.2.3 Automatic Generation of Reference Damage Control Model

The proposed modeling environment automatically generates the reference damage control model for a given graph model. This feature is important because the M&S environment will not be useful for design analyses if the damage control model must be created or modified manually whenever the model configuration changes. As previously described in §4.2.2.1, the reference damage control system is an aggregation of multiple component-level control agents. In order to automate the generation of the reference damage control model, an automation algorithm must be able to create its control agents, and identify the neighbors and the local adjacency matrix *adj_mat* of every control agent. Since the whole purpose of this automation algorithm is to obtain a reference damage control model for every newly reconfigured plant model, the algorithm must be able to extract the topological and other configuration information of the plant model, and use them in order to regenerate a properly working damage control model. Therefore, the developed auto-generation algorithm uses the incidence matrix of the graph-based fluid model as the main input, since the incidence matrix is an extracted form of information of the model's topological connectivity, and the M&S environment is developed to update the incidence matrix according to changes in the topological configuration of the model.

The automation algorithm is constructed of the edge contraction [14, 17] in graph theory and the generation of the edge adjacency matrix with the controller-attached edges of a graph model. The basic steps of the algorithm for auto-generation of the reference damage control model are as follows:

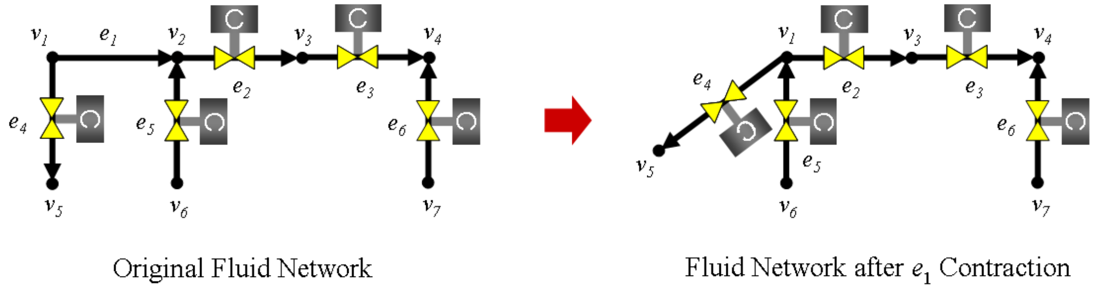
1. Identify all the edges that will not have control units, in a graph model.

2. Create another incidence matrix A_{ce} that is constructed only with the controller-attached edges in the graph model. A_{ce} is obtained by performing edge contractions for all the edges without controllers from the original incidence matrix A of the graph model.

3. Generate the edge adjacency matrix X_{ce} using A_{ce}, the new incidence matrix with the controlled edges.

4. Create controller objects for the controller-attached edges. Identify the neighbors list and the local adjacency matrix of each controller object using A_{ce} and X_{ce}.

The second step in the algorithm, which is the process of edge contraction, comprises two procedures, the deletion of an edge and the fusion of the two incident nodes of the deleted edge. Figure 60 shows an example of the contraction of e_1 in the graph



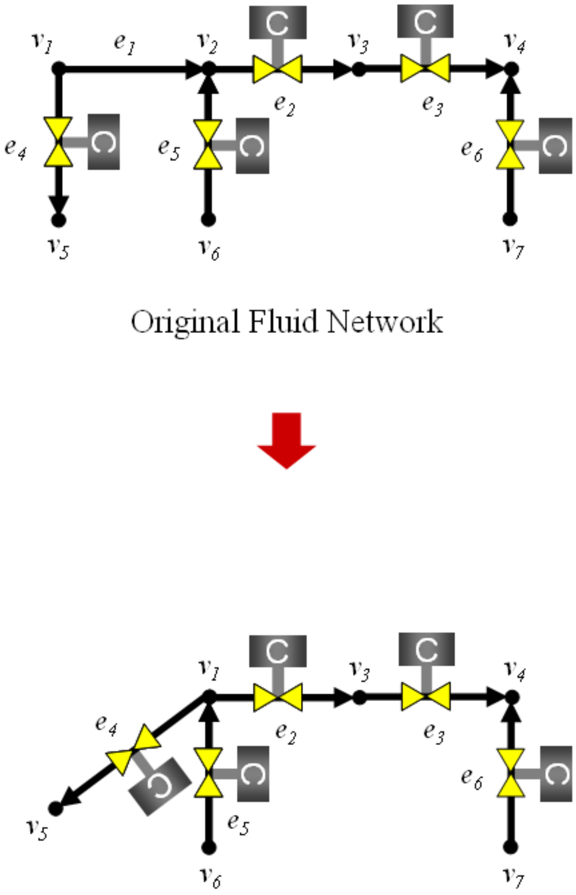
Original Fluid Network

Fluid Network after e_1 Contraction

Figure 60: Contraction of Edge e_1

model of a smart valve-equipped fluid system in Figure 58. As shown in Figure 60, e_1 is removed, and the two previously adjacent nodes v_1 and v_2 are fused as v_1 after the edge contraction. The contraction of e_1 can also be expressed by the manipulation of the incidence matrix. Figure 61 describes the steps of the contraction of e_1, associated with the corresponding incidence matrix manipulation. The step of generating A_{ce} from A using the edge contraction is performed with another three sub-steps. In Figure 61, the first sub-step is equivalent to the identification of the edge to be contracted, and the nodes of the edge. In this step, first, the column representing the edge to be contracted is chosen from the incidence matrix A. Then, the two rows with nonzero elements in this column are selected. These two rows in matrix A represents the two nodes that are adjacent though the edge. As shown in Figure 61, e_1 is the edge to be contracted, so the $1st$ column is selected. Next, the $1st$ and $2nd$ rows of A are selected, since they are the two nonzero elements in the $1st$ column. On the incidence matrix for the first step in Figure 61, these two rows are enclosed with a blue rectangle.

The second sub-step in Figure 61 is equivalent to the fusion of two nodes. The vector summation of the $1st$ and $2nd$ rows of A is performed and the result of the summation replaces the $1st$ row of A. After the summation process, the lst row has the incidence information of the fused node, which is v_1 after the contraction of e_1. Since v_2 is absorbed into v_1, the 2nd row is erased from A.



Original Fluid Network

Fluid Network after e_1 Contraction

1) Find two non-zero elements in col. representing e_1.

$$A = \begin{array}{c|cccccc} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ \hline v_1 & 1 & 0 & 0 & 1 & 0 & 0 \\ v_2 & -1 & 1 & 0 & 0 & -1 & 0 \\ v_3 & 0 & -1 & 1 & 0 & 0 & 0 \\ v_4 & 0 & 0 & -1 & 0 & 0 & -1 \\ v_5 & 0 & 0 & 0 & -1 & 0 & 0 \\ v_6 & 0 & 0 & 0 & 0 & 1 & 0 \\ v_7 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

2) 1^{st} row = 1^{st} row + 2^{nd} row, and eliminate 2^{nd} row.

$$A_{ce} = \begin{array}{c|cccccc} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ \hline v_1 & 0 & 1 & 0 & 1 & -1 & 0 \\ v_3 & 0 & -1 & 1 & 0 & 0 & 0 \\ v_4 & 0 & 0 & -1 & 0 & 0 & -1 \\ v_5 & 0 & 0 & 0 & -1 & 0 & 0 \\ v_6 & 0 & 0 & 0 & 0 & 1 & 0 \\ v_7 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

3) Eliminate 1^{st} column.

$$A_{ce} = \begin{array}{c|ccccc} & e_2 & e_3 & e_4 & e_5 & e_6 \\ \hline v_1 & 1 & 0 & 1 & -1 & 0 \\ v_3 & -1 & 1 & 0 & 0 & 0 \\ v_4 & 0 & -1 & 0 & 0 & -1 \\ v_5 & 0 & 0 & -1 & 0 & 0 \\ v_6 & 0 & 0 & 0 & 1 & 0 \\ v_7 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Figure 61: Manipulation of Incidence Matrix for Contraction of Edge e_1

The third sub-step is equivalent to the deletion of edge e_1. After the second step, the column representing the edge to be deleted must be a zero vector, which means this edge has no connection with the rest of the elements in the graph model, so this column is removed from A.

In the auto-generation algorithm, these three sub-steps of the edge contraction must be iterated for all the edges without controllers. Completion of the iterative edge contraction actually generates the incidence matrix A_{ce}, which has only the controller-attached edges for its columns. In the simple graph model in Figure 60, e_1 is the only edge without a controller, so the algorithm enters the third step, which is the generation of the edge adjacency matrix of the controller attached edges. In order to obtain this matrix, first, the following matrix operation is performed:

$$X_{ce} = A_{ce}^T A_{ce} \tag{87}$$

Next, all the diagonal elements of X_{ce} are replaced with zero values. The resulting new X_{ce} is a symmetric $m \times m$ matrix (where $m = |E(G)|$). This matrix provides the adjacency of the edges of the graph and is very similar to the Laplace matrix in graph theory, which also provides the adjacency of its nodes. If there is a nonzero off-diagonal element, which is $X_{ce}(i, j)$ with $i, j = 1, \dots, m$ and $i \neq j$, then the ith edge is adjacent with the jth edge. For this reason, the author refer to the X_{ce} matrix as the edge adjacency matrix of a graph. Equation (88) is the edge adjacency matrix of the controller-attached edges for the example case in Figures 60 and 61.

$$X_{ce} = \begin{array}{c} \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \end{array} \!\!\! \begin{array}{c} \begin{array}{ccccc} e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{pmatrix} 0 & -1 & 1 & -1 & 0 \\ -1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & -1 & 0 \\ -1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{88}$$

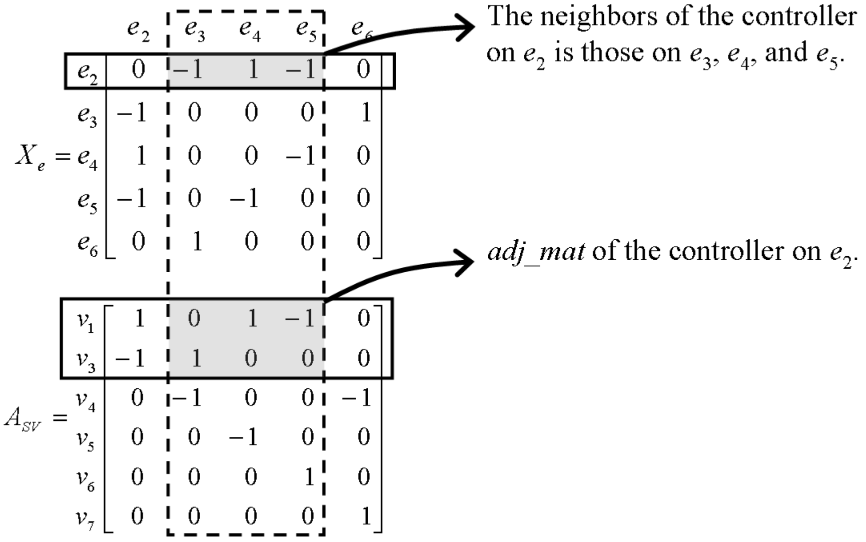


Figure 62: Extraction of Neighbors List and Local Adjacency Matrix (*adj_mat*) of Controller on e_2

In the final step of the algorithm, the neighbors list and the local adjacency matrix of every control unit in the system are extracted from A_{ce} and X_{ce} that were generated from the previous steps. This process is described using the example case for the control unit attached on edge e_2 in Figure 58, which is shown in Figure 62. The process begins from X_{ce} in Equation (88). As previously explained, the nonzero elements of each row or column of this symmetric $m \times m$ matrix represents the corresponding edge's adjacency with the other edges. According to Equation (88), e_2 is represented by the $1st$ row, which is highlighted by the solid rectangle on X_{ce}, in Figure 62. From the $1st$ row, the columns with nonzero elements indicate that e_3, e_4, and e_5, are adjacent with e_2, therefore, the controllers on those edges are the neighbors of the controller on e_2. In Figure 62, the columns for e_3, e_4, and e_5 in matrix X_{ce} are highlighted by the dotted rectangle.

The next step of the process is to find the local adjacency matrix. In order to find it, first, the column representing e_2 is selected from A_{ce}, and then, the two rows with nonzero elements in this column are selected. Between the two rows, the row with -1 in A represents the edges' incidences with the in-node of e_2, and the row with 1 represents their incidences with the out-node of e_2. Since the edges representing the neighboring controllers are already identified using X_{ce}, which is also highlighted by the dotted box on A_{ce} in Figure 62, the matrix elements that are enclosed by both the solid and the dotted boxes of A_{ce} comprises the local adjacency matrix *adj_mat* of the controller on e_2. In Figure 62, these elements are also indicated by the gray color on A_{ce} matrix. This process is iterated for all the controller-attached edges. The complete algorithm for identifying the local adjacency matrix of control units are also described in Algorithm 1.

Algorithm 1 Identify Local Adjacency Matrix

```
$A_{ce}$ = Incidence matrix of the edges with controller objects
$n$ = No. of rows of $A_{ce}$
$m$ = No. of columns of $A_{ce}$ $\{= len\_ce\}$
$C$ = Adjacency matrix of the edges with controller objects
$ce$ = Python list of $m$ controller-attached edges
for $j = 1$ to $m$ do
    for $i = 1$ to $n$ do
        if $A_{ce}[i, j] = 1$ then
            $row1 \leftarrow A_{ce}[i, :]$
        else if $A_{ce}[i, j] = -1$ then
            $row2 \leftarrow A_{ce}[i, :]$
        end if
    end for
    for $i = 1$ to $m$ do
        if $X_{ce}[i, j]$ is not 0 then
            Stack $ce[i]$ in $ce[j].neighbor$
            if $row1$ is not empty then
                Append $row1[i]$ in $ce[j].adj\_mat[1, ]$
            end if
            if $row2$ is not empty then
                Append $row2[i]$ in $ce[j].adj\_mat[2, ]$
            end if
        end if
    end for
end for
```

4.3 Model Integration and Simulation

4.3.1 Model Set-Up

The model setup process (Figure 63) starts with the generation of the library of component-level surrogate models, as discussed in Chapter 3. These component surrogate models are created from an original model that were generated using a domain specific modeling tool, even though it is not shown in the model setup process in Figure 63. Next, a baseline graph model is generated based on the modeler's input file that describes node and edge properties of the graph model. Figure 64 shows how

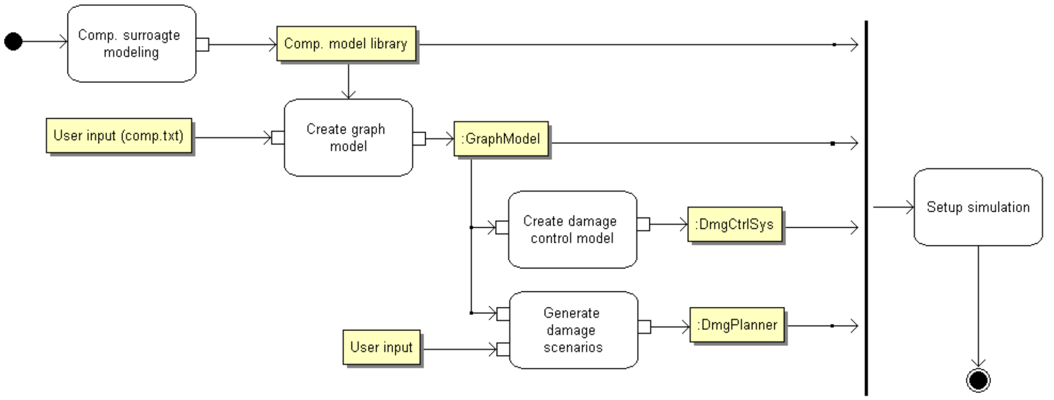
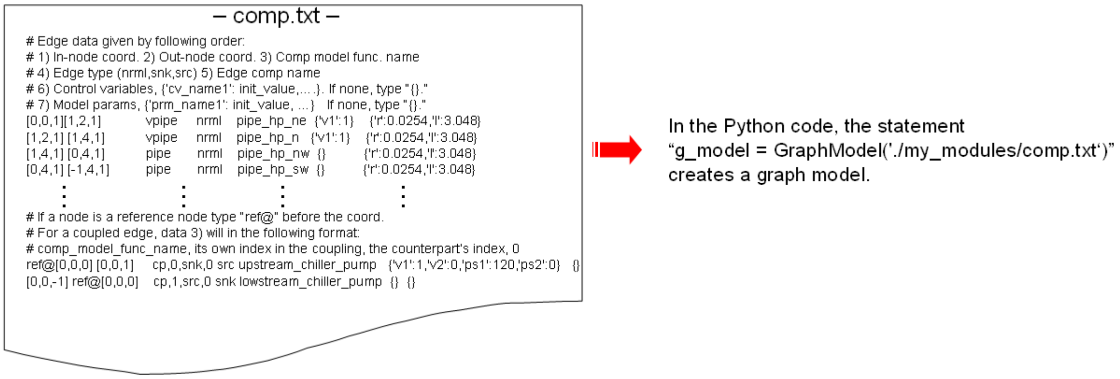


Figure 63: UML Activity Diagram of Model Set-Up Process before Simulation

the model generation was implemented in the Python-based M&S environment. The text file, "comp.txt" in Figure 64 contains the user definition of all the edges such as in and out node coordinates, the name of the component model function, edge type, edge component name, control variables, and model parameters. When a graph model object is created, in a M&S code, by reading the comp.txt file and creating the objects of edges and nodes accordingly. For more complete example of the model initiation input file, see Appendix B.1.1 Figure 65 describes a simplified process flow about the generation of the component surrogate models and the graph-based model. Based on an original model developed in an domain-specific modeling tool, a modeling engineer create the sketch or conceptual map of the graph representation of the



```
– comp.txt –
# Edge data given by following order:
# 1) In-node coord. 2) Out-node coord. 3) Comp model func. name
# 4) Edge type (nrml,snk,src) 5) Edge comp name
# 6) Control variables, {'cv_name1': init_value,... .}. If none, type "{}."
# 7) Model params, {'prm_name1': init_value, ...}   If none, type "{}."
[0,0,1][1,2,1]         vpipe    nrml    pipe_hp_ne  {'v1':1}    {'r':0.0254,'l':3.048}
[1,2,1] [1,4,1]        vpipe    nrml    pipe_hp_n   {'v1':1}    {'r':0.0254,'l':3.048}
[1,4,1] [0,4,1]        pipe     nrml    pipe_hp_nw  {}          {'r':0.0254,'l':3.048}
[0,4,1] [-1,4,1]       pipe     nrml    pipe_hp_sw  {}          {'r':0.0254,'l':3.048}
     ⋮                   ⋮                ⋮                        ⋮
# If a node is a reference node type "ref@" before the coord.
# For a coupled edge, data 3) will in the following format:
# comp_model_func_name, its own index in the coupling, the counterpart's index, 0
ref@[0,0,0] [0,0,1]    cp,0,snk,0 src upstream_chiller_pump   {'v1':1,'v2':0,'ps1':120,'ps2':0}   {}
[0,0,-1] ref@[0,0,0]   cp,1,src,0 snk lowstream_chiller_pump  {}  {}
```

In the Python code, the statement
"g_model = GraphModel('./my_modules/comp.txt')"
creates a graph model.

Figure 64: User Inputs for Initializing Graph Model

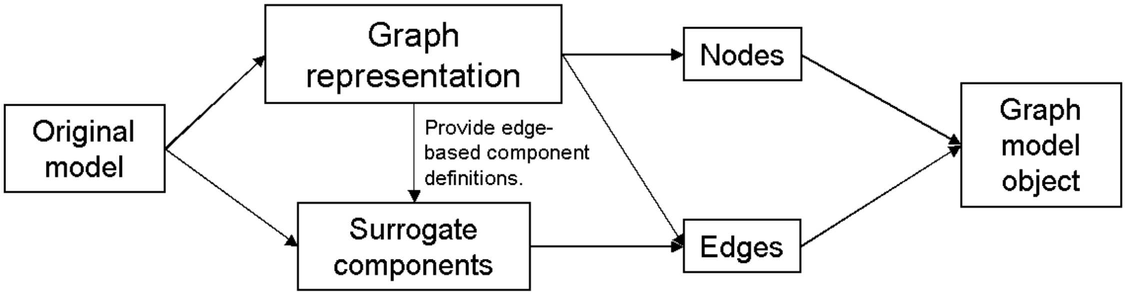


Figure 65: Simplified Diagram of Graph-Based Model Generation Process Flow

original model. Edges of the graph representation are grouped by the configurational commonality of them, and for each group of edges, regressors and outputs of a component surrogate model are defined. Then, the component surrogate model is built, which becomes the common component behavioral model for the edges of the group. With the component surrogate models and the graph representation, node and edges objects are implemented, and these objects form the graph-based model object.

Once the graph model object is created, it becomes the input for generating two more model objects for a damage analysis. These are the damage planner and the reference damage control model objects. The damage planner object has a list of damage scenarios that are defined by different values of the center point and the radius of a damage bubble. During each simulation, a damage planner reads a set of the properties for a damage bubble in the list in order, and creates a damage bubble

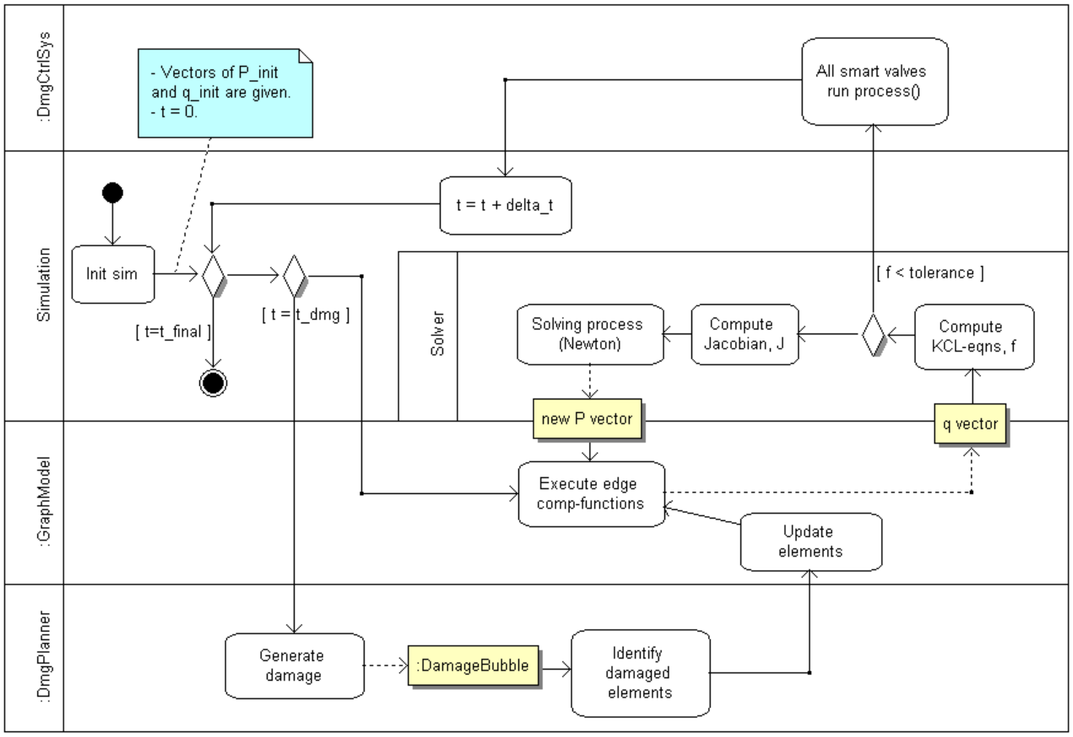


Figure 66: Simulation Process

object when damage is triggered at simulation time $t = t_{dmg}$, which is set by a user. The current damage planner object generates only one damage bubble per simulation run, so it needs to be modified if more bubbles are necessary. Also, based on the algorithm in §4.2.3, the reference damage control model is generated automatically for a given graph model.

4.3.2 Simulation Process and Jacobian Computation

The model objects and the component model library then enter the simulation process. The UML activity diagram in Figure 66 shows a simplified process for a single simulation run. The swim-lane partitions in Figure 66 represent the process separations of different objects. The simulation begins with initializing the pressures at all the nodes and the flow rates on all the edges in the graph-based model. For each discrete simulation time step, the estimate of $\vec{q}_t$, which is the vector of flow rates of the edges for the current time step, is computed by executing the component behavioral models connected to all the edges in the graph model object. These model functions are all in the form of $q_t = f_{NN}(q_{t-1}, \Delta P_t, \ldots)$, and since the previous values of the node pressures $\vec{P}_t$ is unknown yet, $\vec{P}_{t-1}$ is used as the estimate of $\vec{P}_t$ for the initial computation of $\vec{q}_t$. These initial estimate of flow rates $\vec{q}_t$ would not necessarily satisfy the KCL constraints given by the connection topology of the graph-based model, a nonlinear iterative solution process is performed for each time step, in order to find $\vec{P}_t$ that produces $\vec{q}_t$ satisfying the KCL constraints. Since this iterative solution process is the core of the simulation process, the details of the solution algorithm is given in Algorithm 2, which uses Newton method as the nonlinear solver. As described in line 2 in Algorithm 2, the solver begins its process with the node pressures and the edge flow rates from the previous simulation time step as the input, but in the very beginning of the simulation, $\vec{P}_{t-1} = \vec{P}_0$ and $\vec{q}_{t-1} = \vec{q}_0$.

When the simulation triggers damage, which is scheduled at simulation time t_{dmg}, the simulation executes the additional process of creating and applying a damage bubble object to the graph-based model. The process in the damage bubble object identifies and modifies, or removes, the damaged components of the graph model. As a result, the simulation has a new model that was reconfigured by the damage bubble object. With this damaged model, the simulation performs the same routine of iterative solution process of finding $\vec{P}_t$ and $\vec{q}_t$ as previously described. The damaged model also induces the reaction of the reference damage control model for identifying and isolating the damage in the graph-based model.

In Algorithm 2, the volumetric flow rate models are functions of pressure, the goal of the solver is to find the correct $\vec{P}_t$ values that produce the $\vec{q}_t$ satisfying $A\vec{q}_t(\vec{P}_t) = 0$, in other words, the flow conservation (or KCL) constraints imposed by the network connection topology. This requires an iterative solution method, because the component model of each edge, which computes q_t, is a nonlinear function. Algorithm 2 uses the Newton method in which Jacobian J of the vector h, the flow sums at the

Algorithm 2 Solution Process

```
GIVEN: $g\_model : GraphModel$ {graph model object}
GIVEN: $P_{t-1}$, $q_{t-1}$ {Type: array. Previous values of pressures and flow rates}

$N_n = no.\ of\ nodes$
$N_e = no.\ of\ edges$
$node = g\_model.node$
$edge = g\_model.edge$
$P_t = P_{t-1}$ {Use $P_{t-1}$ as the first guess of $P_t$}
for $i = 1$ to $N_e$ do
    $edge[i].q_{t-1} = q_{t-1}[i]$
end for
loop
    for $i = 1$ to $N_n$ do
        $node[i].np_t = P_t[i]$ {$np_t$ = node pressure at time $t$}
    end for
    $A = g\_model.get\_A()$ {$A$: incidence matrix}
    $q_t = g\_model.get\_q_t()$ {Execute $edge.model(q(t-1), \Delta P(t), cv(t), prm)$ for
    all edges}
    $h = A \cdot q_t$
    if $\|h\| \leq \epsilon$ then
        Escape loop
    end if
    $J = jacobian()$
    $P_t = P_t - J^{-1}h$
end loop
return $P_t$, $q_t$
```

nodes, which represent the KCL constraints, can be computed using the available topological information of the system. For convenience, $\vec{P_t} = \vec{P}$ and $\vec{q_t} = \vec{q}$ in the following derivation.

The Jacobian of vector $h(\vec{P(t)})$ is given by

$$J(i,j) = \frac{\partial h_i}{\partial P_j} \tag{89}$$

where i and j are integers such that $1 \leq i, j \leq p$ of the graph G, and p is the number of normal nodes. Also recalling that, $h_i = A_i \cdot \vec{q}(\vec{P})$ where A_i denotes the ith row of A, and $\vec{q}(\vec{P})$ is the $|E(G)|$ length vector of edge flow rates at time t, the Jacobian is given as

$$\begin{aligned} J(i,j) &= \tfrac{\partial}{\partial P_j} A_i \vec{q} \\ &= A_i \tfrac{\partial \vec{q}}{\partial P_j} \\ &= A_i \begin{bmatrix} \tfrac{\partial q_1}{\partial P_j} & \tfrac{\partial q_2}{\partial P_j} & \cdots & \tfrac{\partial q_k}{\partial P_j} & \cdots & \tfrac{\partial q_m}{\partial P_j} \end{bmatrix}^T \quad (m = |E(G)|) \end{aligned} \tag{90}$$

In fact, the flow rate q_k of the kth edge takes the pressure difference as an input, which can be expressed as

$$q_k(\Delta P) = q_k(P_{in} - P_{out}) \tag{91}$$

where P_{in} and P_{out} are the pressure at the in and out nodes of the kth edge. By the chain rule, this leads to three different cases in the calculation of $\dfrac{\partial q_k}{\partial P_j}$ which are

$$\frac{\partial q_k}{\partial P_j} = \begin{cases} \frac{\partial q_k}{\partial \Delta P} & \text{if } P_{in} = P_j \\ -\frac{\partial q_k}{\partial \Delta P} & \text{if } P_{out} = P_j \\ 0 & \text{otherwise} \end{cases} \tag{92}$$

With a careful observation of A_j, it is already known that:

$$a_{jk} = \begin{cases} 1 & \text{if } k\text{th edge has a flow out of } j\text{th node} \\ -1 & \text{if } k\text{th edge has a flow into } j\text{th node} \\ 0 & \text{otherwise} \end{cases} \tag{93}$$

By associating this observation of Equation (93), Equation (92) can be expressed more simply as,

$$\frac{\partial q_k}{\partial P_j} = a_{jk} \frac{\partial q_k}{\partial \Delta P} \tag{94}$$

Now plugging Equation (94) into Equation (90), the entire J matrix is,

$$J = A \cdot \begin{bmatrix} a_{11}\dfrac{\partial q_1}{\partial \Delta P} & a_{21}\dfrac{\partial q_1}{\partial \Delta P} & \cdots & a_{n1}\dfrac{\partial q_1}{\partial \Delta P} \\ a_{12}\dfrac{\partial q_2}{\partial \Delta P} & a_{22}\dfrac{\partial q_2}{\partial \Delta P} & \cdots & a_{n2}\dfrac{\partial q_2}{\partial \Delta P} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m}\dfrac{\partial q_m}{\partial \Delta P} & a_{2m}\dfrac{\partial q_m}{\partial \Delta P} & \cdots & a_{nm}\dfrac{\partial q_m}{\partial \Delta P} \end{bmatrix} = A \cdot \begin{bmatrix} \dfrac{\partial q_1}{\partial \Delta P} \cdot A_1^T \\ \dfrac{\partial q_2}{\partial \Delta P} \cdot A_2^T \\ \vdots \\ \dfrac{\partial q_m}{\partial \Delta P} \cdot A_m^T \end{bmatrix} \tag{95}$$

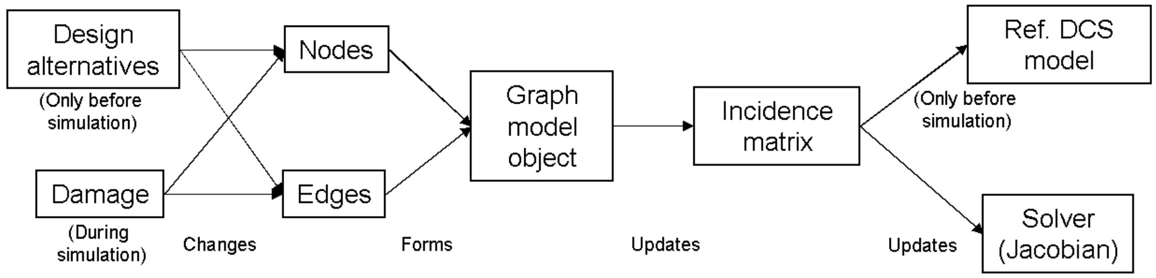


Figure 67: Interactions of Elements Model Reconfiguration

So the Jacobian, J can be computed with the derivatives of the m edge model functions and the incidence matrix A. The numerical implementations of both the iterative solver and Jacobian algorithms can be found in Appendix B.3.3.

The graph-based model in the M&S environment can be changed during simulation by damage on the system, or before simulation for model reconfiguration according to design changes of the system. With a change of the graph-based model, the Jacobian matrix in the simulation environment and the reference damage control model are automatically regenerated. Figure 67 is a simplified diagram describing the interactions between the elements in the M&S environment when model reconfigurations occur.

CHAPTER V

IMPLEMENTATION EXAMPLE

5.1 Brief Introduction of Chilled-Water Model of Notional Ship

The demonstration model in Figure 69 is basically a scaled-down version of CW-RSAD model that was created by NSWC-CD [39, 66]. CW-RSAD is a 1/4 scaled-down physical model of about half of the chilled-water system of the notional DDG-51 class destroyer. The demonstration model for this research was named the notional-YP (Yard patrol craft) fluid system model, since its geometric or spatial layout mimicked that of YP-676, which is used as a training craft in the U.S. Naval Academy [6, 21]. As a part of the ONR-funded IRIS (Intelligent Reconfigurable Integrated Systems) project in Aerospace Design Laboratory of Georgia Institute of Technology, the YP fluid model was created using Flowmaster®V7, a 1-D pipeline M&S tool, which

Figure 68: YP-676 Yard-Patrol Craft

has also been used as a domain M&S tool in the NSWC-CD's integrated simulation environment for damage propagation analysis. The system configuration consists of a pair of redundant pump-chiller sub-networks and seven heat exchanger units for serving six thermal service loads. The heat exchanger units HEX no.4 and HEX no.5 in Figure 69 are a redundant pair serving a single thermal service load, and each heat exchanger has either one or two (as redundancy) flow control valves in it. Similarly, each pump-chiller sub-network contains two pumps as redundancy and a reservoir for an extra water resource in an emergency. The system has 18 smart valves for automated damage isolation.

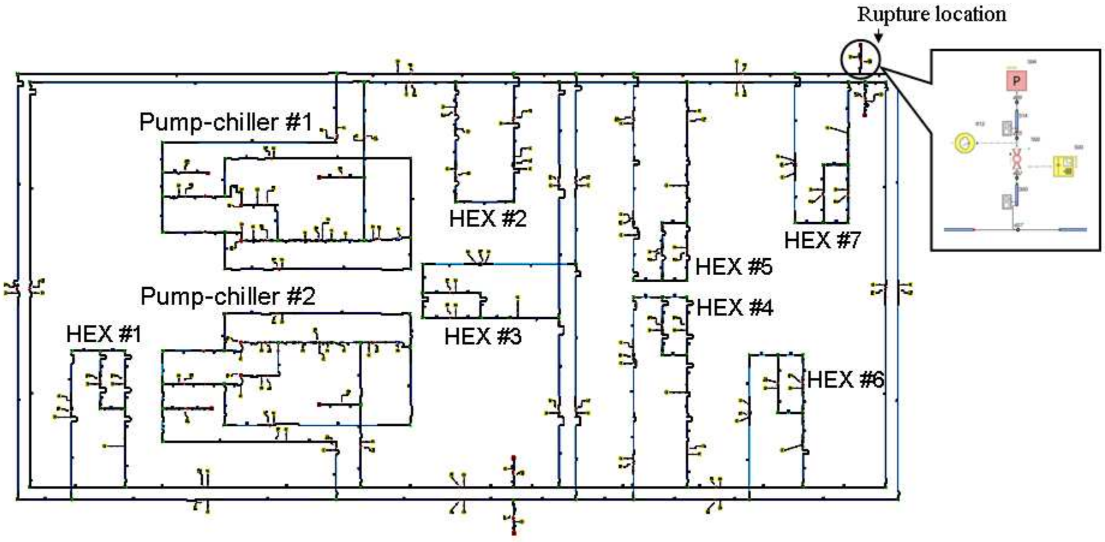


Figure 69: Chilled-Water Cooling Model of Notional YP and Rupture Location

Figure 69 shows how damage modeling was implemented in the Flowmaster model. Applying the Navy's approach that was introduced in §1.3.1, a rupture valve, which is shown on the upper right corner of Figure 69, was created at a location at which a damage was scheduled to occur during a simulation.

5.2 Graph-Based Surrogate Model for Notion YP Fluid System

5.2.1 Graph-Based Representation

The first step was the translation of the notional-YP fluid model into a graph-based representation, such as the one given in Figure 70. In this representation, the model has 52 edge components and 40 nodes, but all the components belong to one of five different component types. In other words, all the edge components can be represented by five component models with appropriate choices of model parameters for higher reusability. Table 14 is the list of the component models defined in the graph-based model representation, and the initial values of control variables and parameters of the 52 edge components can be found from the graph-model initialization input file in Appendix B.1.1.

As in Table 14, the PC model (pump-chiller model) has two flow rates and two node pressures as the part of its inputs and is represented by two inter-coupled edges in the graph model as explained in §4.1.2.1. Although not shown in the Figure 70, each PC edge component are constructed of two edges, which are connected by a reference node. For modeling simplicity, the reservoirs are set to provide unlimited water resources, but in reality, they have only a finite amount of the water and would be depleted in a short time, if a rupture occurs in the system, but there is no reaction of damage isolation. As described in §4.1.2.3, the nodes were defined by their topological coordinates.

5.2.2 Generation of Component Surrogate Models

Seven NN-surrogate models were created to model the five different types of components described in Table 14. Table 15 shows the specification of the NN-surrogate models. In the numerical implementation, these component models are all defined in

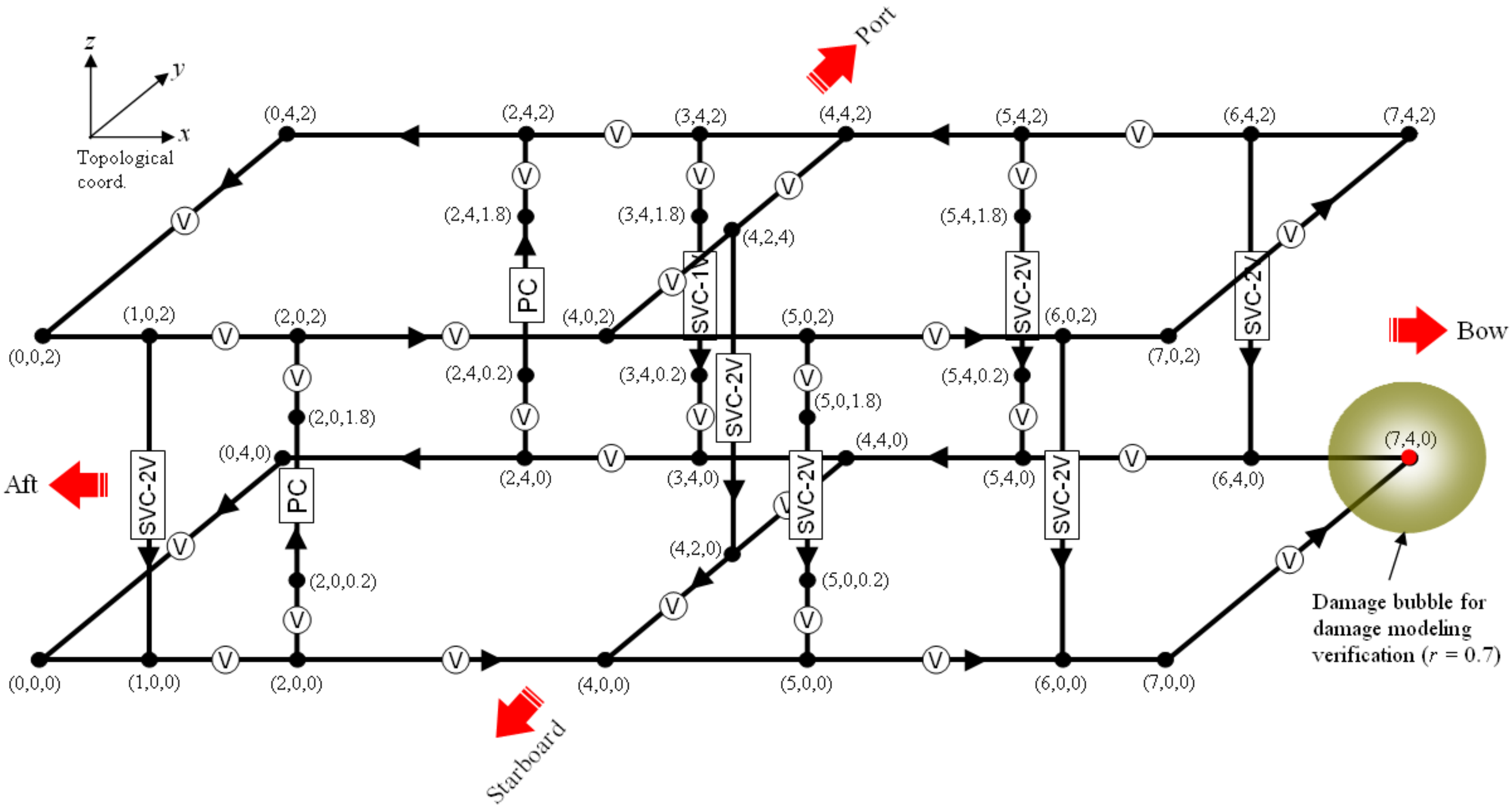


Figure 70: Graph Representation of YP Fluid Model

Table 14: Edge Component Types

Name	Component type	No. of comps	State vars	Boundary cond.	Control vars	Parameters
PIPE	Simple pipe	14	q (m^3/s)	ΔP (Pa)	None	l (length,m)
VPIPE	Pipe with a valve	30	q	ΔP	v_1 (0 to 1)	l, d (diameter,m)
SVC-1V	Svc load with a valve	1	q	ΔP	v_1	None
SVC-2V	Svc load with two valves	6	q	ΔP	v_1, v_2	None
PC	Pump-chiller sub-net	2	q_{in}, q_{out}	P_{in}, P_{out}	v_1, ω_{pmp} (rad/s)	None

components.py file, which is given in Appendix B.3.2 and works as the component-model library.

First, the component model was built using the Flowmaster® tool and connected to Matlab® using a COM interface in order to perform the computer experiments. Then the training data was created according to the settings and process in §3.3. For each surrogate model, the training was performed five times with the same training data, and then the NN model with the smallest training MSE was chosen as the final model.

Particularly for the VPIPE model, four different NN-surrogate models were created to represent it. This was done to achieve better accuracy and efficiency in modeling. During the several trial-and-error iterations for creating a surrogate model of the VPIPE model, the inclusion of both pipe length and diameter into the input space turned out to be a less suitable decision since the response data from the computer experiment came with a very large order of magnitude, which deteriorated the accuracy of the resulting surrogate model. In fact, the YP fluid system was made of pipes with only three different diameters so the approach of creating a separate surrogate model for each pipe diameter resulted in a good accuracy without sacrificing affordability of modeling.

Although most of the surrogate models had relatively large training data, the computational cost for both computer experiments and training the models were not considerably high, except for the PC model. The computing hardware had an Intel Core II Duo processor and 2 Gbyte memory, and for all the models other than the PC model, the computational time for the computer experiment was well less than one hour and a single training took only a few minutes. On the other hand, the PC model, which had the most complicated configuration among the component models, needed significantly longer time – about 4 to 5 hours of computing time – than others.

While testing the surrogate models with valves, a common problem was that their

Table 15: NN-Surrogate Models Specification

Name	NN-functions	Input range	Data size	No. of neurons	Training MSE	Test MSE
PIPE	pipe	$-0.001 \leq q \leq 0.001$ $-10^4 \leq \Delta P \leq 10^4$ $0.5 \leq l \leq 5$	$21,511$	6	1.1351×10^{-4}	3.1619×10^{-4}
VPIPE	vpipe_0127	$d = 0.0127$ $0.3 \leq l \leq 1.5$ $-0.001 \leq q \leq 0.001$ $-10^5 \leq \Delta P \leq 10^5$ $0 \leq v_1 \leq 1$	$37,817$	8	8.3003×10^{-4}	1.7213×10^{-3}
	vpipe_01905	$d = 0.01905$ $0.3 \leq l \leq 6.5$	$102,285$	8	1.9484×10^{-3}	2.9164×10^{-3}
	vpipe_0254_sdp	$d = 0.0254$ $2 \leq l \leq 7$ $0 \leq v_1 \leq 1$ $-0.001 \leq q \leq 0.001$ $-4 \times 10^3 \leq \Delta P \leq 3 \times 10^3$	$26,694$	8	1.0910×10^{-5}	4.9561×10^{-5}
	vpipe_0254	$-0.0005 \leq q \leq 0.0005$ $-7 \times 10^4 \leq \Delta P \leq 7 \times 10^4$	$29,271$	16	2.0953×10^{-5}	3.1013×10^{-3}
SVC-1v	svc_1v	$-0.0005 \leq q \leq 0.0005$ $-1 \times 10^5 \leq \Delta P \leq 1 \times 10^5$ $0 \leq v_1 \leq 1$	$10,077$	7	1.1351×10^{-4}	3.1619×10^{-4}
SVC-2v	svc_2v	$-0.0003 \leq q \leq 0.0003$ $-5 \times 10^4 \leq \Delta P \leq 1 \times 10^5$ $0 \leq v_1 \leq 1$	$10,077$	11	1.8831×10^{-5}	4.5144×10^{-5}
PC	pc	$-0.001 \leq q_{in}, q_{out} \leq 0.002$ $10^5 \leq P_{in} \leq 2.5 \times 10^5$, $10^5 \leq P_{out} \leq 4.5 \times 10^5$ $0 \leq v_1 \leq 1$, $0 \leq \omega_{pmp} \leq 400$	$13,200$	11	5.7994×10^{-4}	9.3825×10^{-4}

accuracy was insufficient when the valve input was close to zero (i.e., a complete closure). To address this problem, the following scheme of error correction was applied:

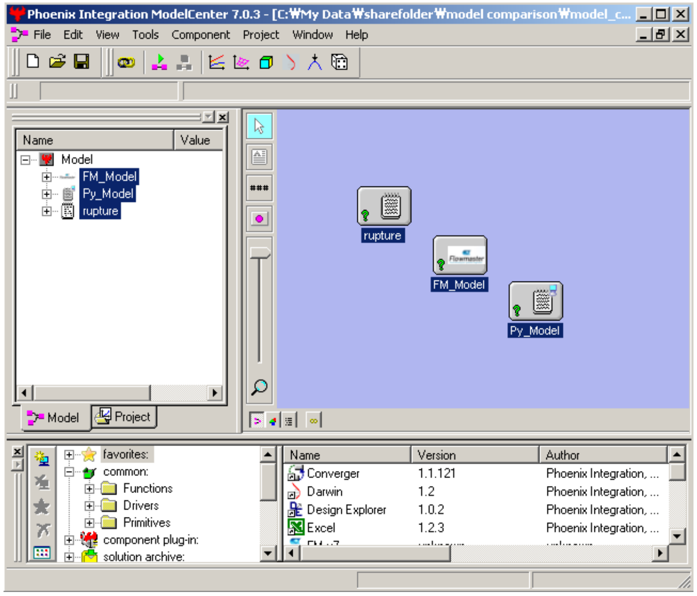
$$q_t = \begin{cases} f_{NN}\left(q_{t-1}, \Delta P, v, \ldots\right) & \text{, if } v > \epsilon \\ e^{a(v-\epsilon)} f_{NN}\left(q_{t-1}, \Delta P, v, \ldots\right) & \text{, if } v \leq \epsilon \end{cases} \tag{96}$$

where, ϵ is a threshold of valve opening value that is close to zero. This scheme works as follows: a decimal fraction value multiplies the surrogate model output to reduce its magnitude when the valve input is close to zero. For this particular case, the coefficient a was set in the range of 17 to 35 and the valve input threshold ϵ was set to 0.05, which gave the effect of multiplying some value within 0.2 to 0.4 to the model output when the valve was completely closed.

5.3 Simulation-Based Design Analysis

All simulations ran with a discrete time step of 0.05 second, and as the initial setting of the simulations, the PC no.1 sub-net operated with a single pump turned on with a fixed speed of 200 rad/s (about 1910 RPM), while PC no.2 was in off-state. When PC no.1 was damaged, the controller on PC no.2 turned the pump in the PC no.2 on in order to continue the system-level operation. The 18 damage control valves, which were placed throughout the two looped pipelines and their by-pass lines, were connected to the reference damage control model created by the algorithm in §4.2.3.

Three analyses were included in this demonstration. The first was the verification of the graph-based surrogate model by performing an open-loop damage simulation. In the first analysis, the simulations of both the Flowmaster and the graph-based models of the YP fluid system were executed with no damage control attached. As the damage scenario for both models, a rupture occurred at the two second point of the simulation with the location on (7,4,0) in Figure 70. Next, the accuracy and the computational cost of the graph-based surrogate model were compared to the Flowmaster counterpart, based on the simulation results from the two models. As an additional setting to mimic the condition that the two models were run under an integrating framework, both the Flowmaster and the graph-based surrogate models were linked to *ModelCenter*® 7.0 using its script-wrapping support, as shown in Figure 71.



Name	Version	Author
Converger	1.1.121	Phoenix Integration, ...
Darwin	1.2	Phoenix Integration, ...
Design Explorer	1.0.2	Phoenix Integration, ...
Excel	1.2.3	Phoenix Integration, ...

Figure 71: ModelCenter®7.0 Environment

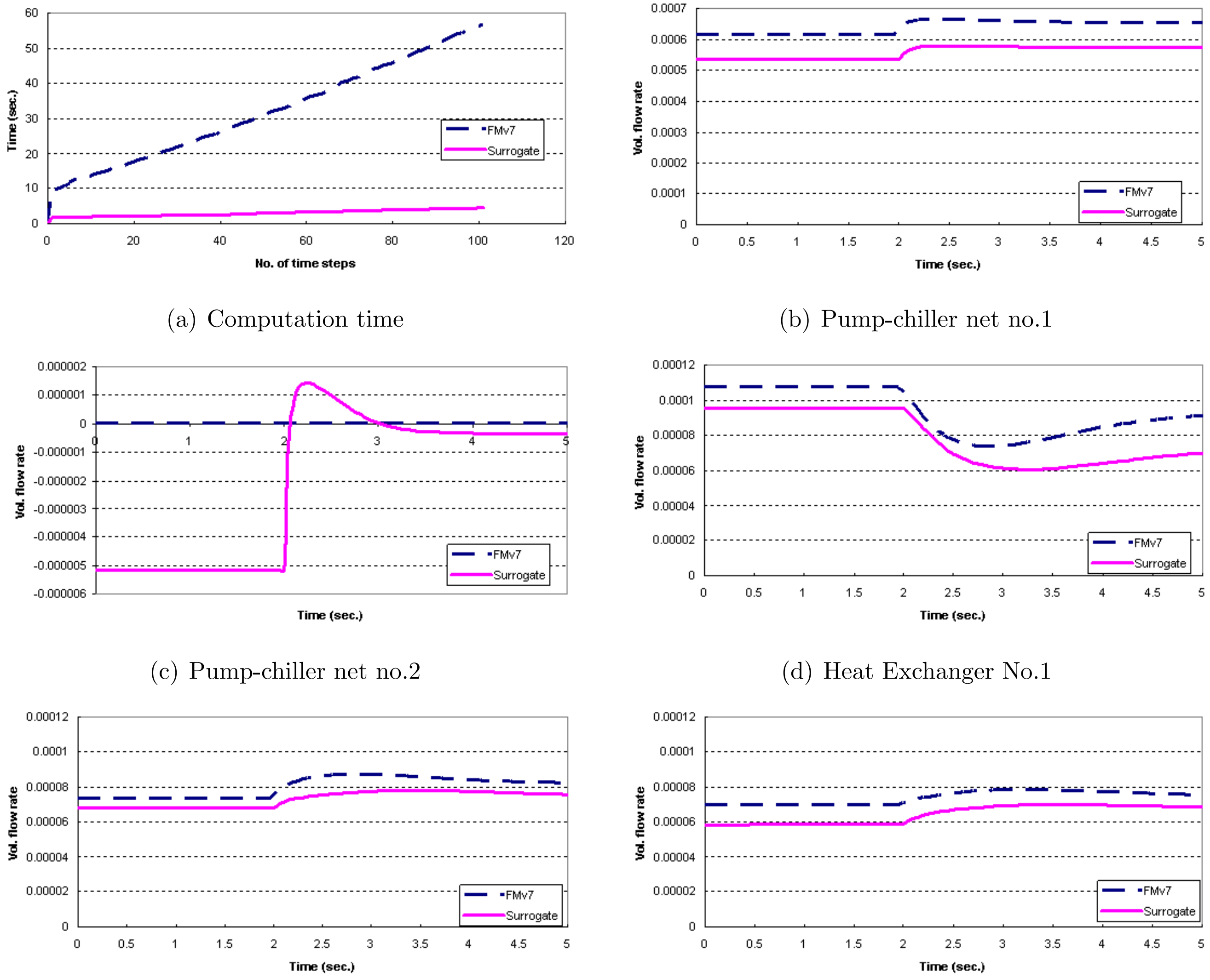
As the second analysis, a simulation-based damage experiment was performed with 28 different damage locations. For all simulation runs, a damage was set to be triggered at the one second point of the simulation. A damage bubble representing the damage was set to the same radius of 0.7 for all simulations, and the 28 damage locations were uniformly distributed through the system and are given in Figure 76(b). For each simulation run, the operational recovery capability of the fluid system was quantified and plotted, as an example of how this model may help the system engineers explore and build intuition for designing more resilient systems.

In the last analysis, the design alternatives with different smart-valve placement were generated from the original fluid system in Figure 70, and the damage experiments were performed for them to find out the designs that delivers better recovery capability than the original system.

5.3.1 Model Verification

The analysis was performed with the simulation running for 5 seconds of simulation time, and the response comparison is shown in Figures 72 and 73. As in Figure 72(a), the graph-based surrogate model took 4.48 secs which was 12.6 times faster than the Flowmaster model, which took 56.48 secs. However, as a compromise to the benefit in the computational cost, the graph-based surrogate model contained errors and biases in its response compared to the responses of the Flowmaster model.

These errors and biases could come from a combination of various reasons, but the highest contributor is more likely the insufficient model accuracy of the PC surrogate model. The PC Flowmaster model had very slower dynamics than other models, and needed significantly longer simulation time to reach its steady state than the other component surrogate models, resulting in the excessively large size and generation time of the raw data (time-series responses) from the computer simulations using it. As a way to mitigate these problems, when the two-stage experimental design (see §3.3.1) was built and executed for the PC model, only 30% of the original second-stage DOE were chosen randomly and used as the scenario at each simulation. Also, for the computer experiment, a maximum simulation time was set for all the simulation runs so the simulations stop even if they did not reach the steady-state responses, in order to keep the raw data from being excessively large to manage. With these two settings, the computational time for generating the training data from the PC Flowmaster model was about 4 to 5 hours as previously mentioned in §5.2.2. It is significantly faster than the computation with the full, original size of the DOE with dynamic variables and no final-time threshold during simulation, which may take a few days to be completed. However, the PC surrogate model was created from insufficiently large enough training data set for its dimension and size of input space, so this led to higher prediction errors for the PC surrogate model when in use.



(a) Computation time

(b) Pump-chiller net no.1

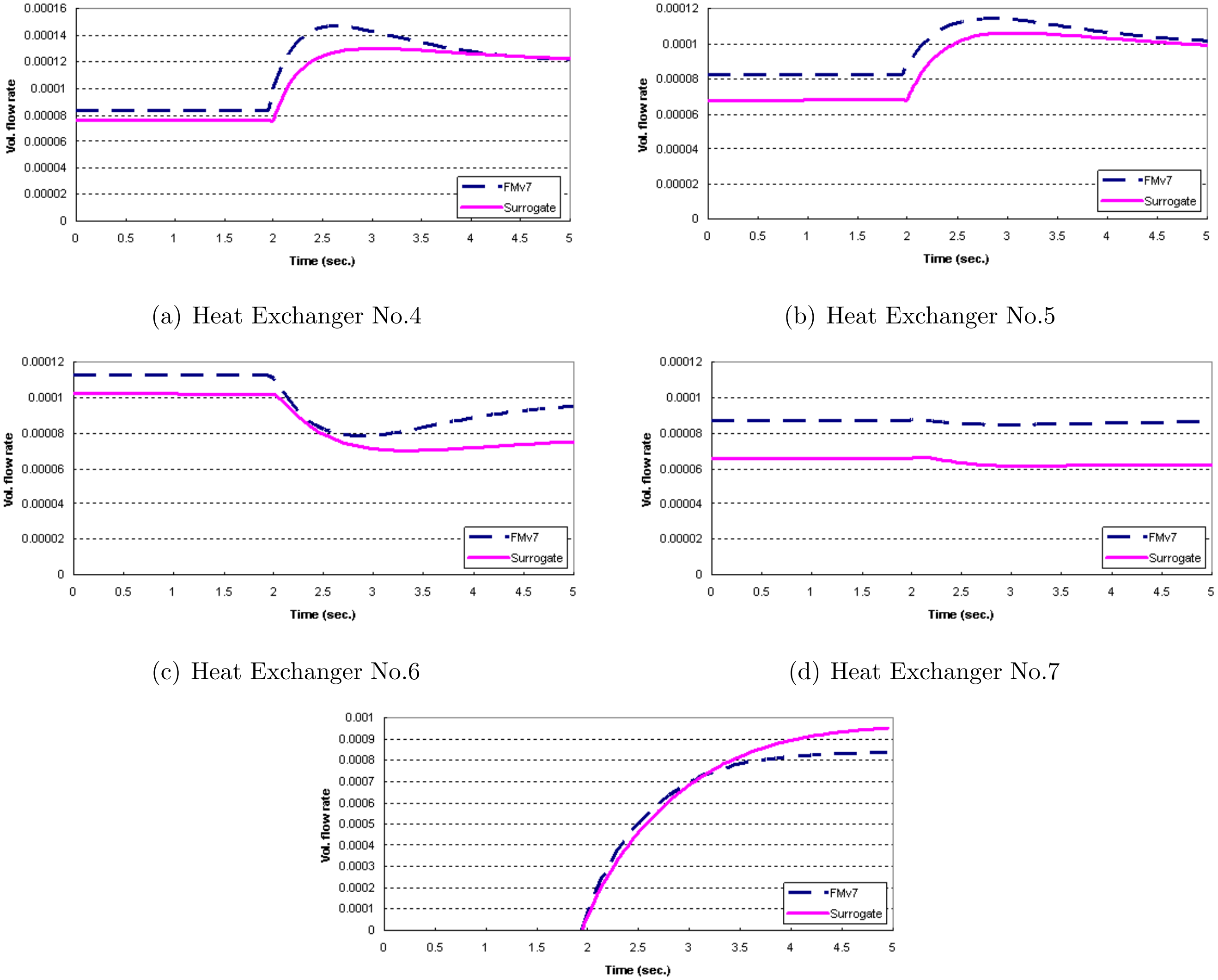
(c) Pump-chiller net no.2

(d) Heat Exchanger No.1

(e) Heat Exchanger No.2

(f) Heat Exchanger No.3

Figure 72: Comparison of the Responses of Flowmaster and Graph-Based Surrogate Model with the Rupture at (7,4,0), Part 1



(a) Heat Exchanger No.4

(b) Heat Exchanger No.5

(c) Heat Exchanger No.6

(d) Heat Exchanger No.7

(e) Total Rupture Flow

Figure 73: Comparison of the Responses of Flowmaster and Graph-Based Surrogate Model with the Rupture at (7,4,0), Part 2

Despite this problems, the graph-based surrogate model is still very useful, especially for the early phase of the design process. In practice, the Flowmaster model is not necessarily more accurate than the graph-based surrogate model when it comes to modeling in the early design stage because a large portion of the system detail is still unknown or undecided. Instead, what is needed more in the early design stage is a computationally affordable model that lets a designer run a large number of analysis cases for exploring as large a design space as possible, meaning that the graph-based model can be an attractive choice to serve the purpose.

5.3.2 Damage Analysis of Notional YP Fluid Model

In the second analysis, the system was closed by the reference damage control loop (see §4.2.2), and each of the 28 simulations ran for 10 seconds of simulation time. Figure 74 shows the different locations of the centers of the damage bubbles in the 28 simulation cases.

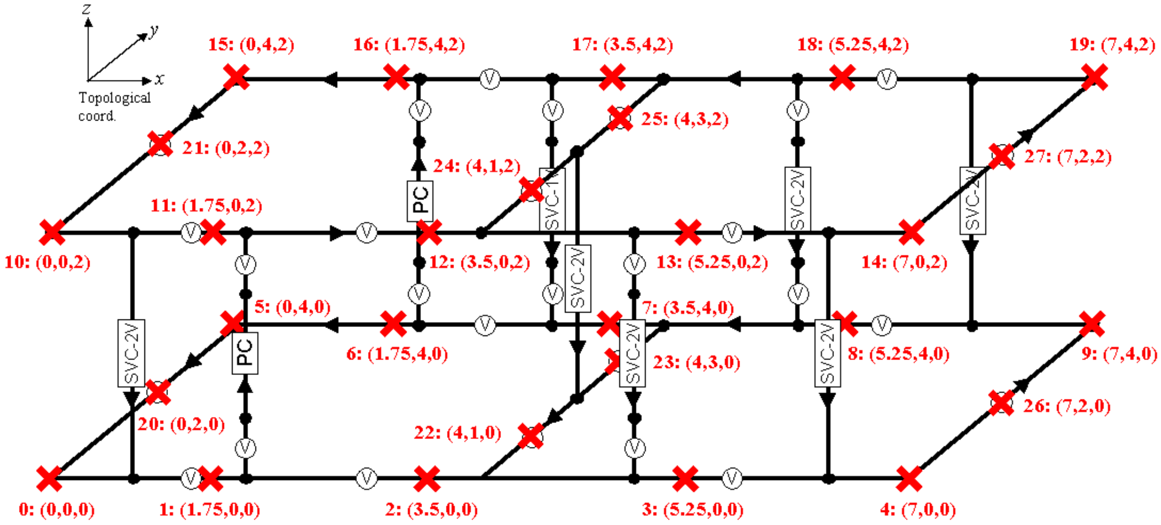
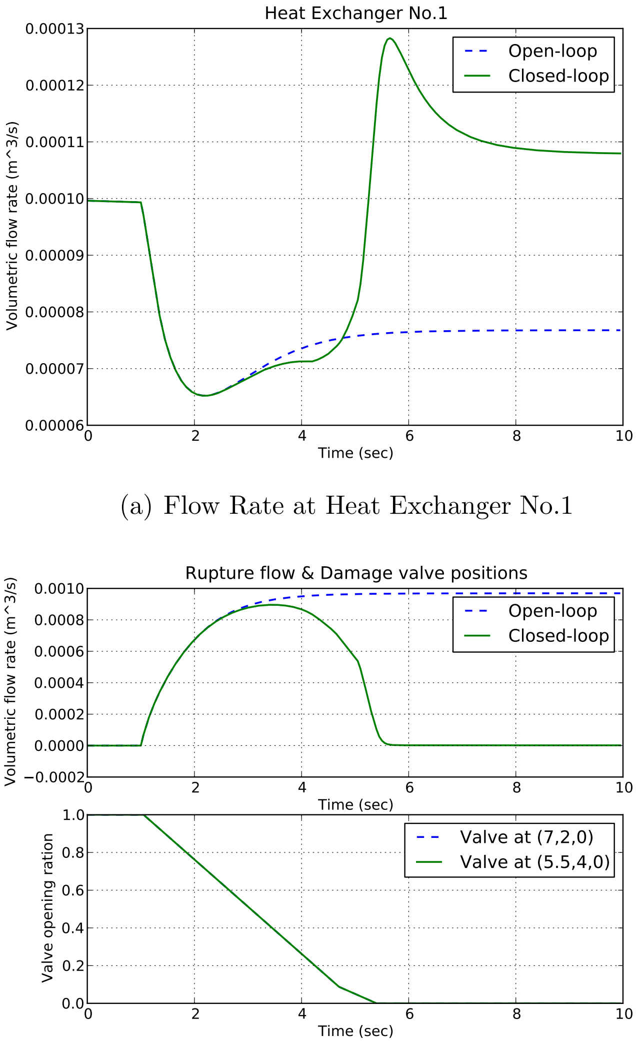


Figure 74: Damage Locations for 28 Simulation Cases

Figure 75 shows the comparison of two of the open-loop and the closed-loop responses for the graph-based surrogate model with the same damage condition as the one in the first analysis. In Figure 75(a), right after the rupture occurred at the one



(a) Flow Rate at Heat Exchanger No.1

(b) Total Rupture Flow and Damage Control Valve Inputs

Figure 75: Comparison of Open-Loop and Closed-Loop Responses of YP-Fluid System with Rupture at (7,4,0)

second point of the simulation, the closed-loop system had a similar sudden drop in flow rate as in the open-loop system, but eventually settled into a new recovered steady state due to the successful isolation of the rupture. Although Figure 75 is only one of the 28 results, the other behaviors are similar to Figure 75.

Figure 75(a) reveals an interesting aspect of the recovered steady state of the system, which is that the flow rates at the heat exchanger units are usually increased after the system recovers from a rupture. It happens when the closure of damage control valves isolates not only a rupture, but also a heat exchanger unit near the rupture from the rest of the system, so the total number of working heat exchangers that share the chilled-water resource provided by the PC component decreases. As a result, the amount of the chilled water available for each heat exchanger increases after the system is recovered.

5.3.2.1 Defining Operation Capability Rate of YP-Fluid System

In order to quantify and measure the system recovery performance for every simulation case, the operation capability rate (OCR) of the YP-fluid system was defined in the following way:

$$OCR = \frac{w_1\tilde{q}_1^f + w_2\tilde{q}_2^f + w_3\tilde{q}_3^f + w_4 \cdot max\left(\tilde{q}_4^f, \tilde{q}_5^f\right) + w_5\tilde{q}_6^f + w_6\tilde{q}_7^f}{w_1 q_1^o + w_2 q_2^o + w_3 q_3^o + w_4 \cdot max\left(q_4^o, q_5^o\right) + w_5 q_6^o + w_6 q_7^o} \tag{97}$$

and

$$\tilde{q}_i^f = \begin{cases} q_i^f & \text{, if } q_i^f \leq q_i^o \\ q_i^o & \text{, otherwise} \end{cases} \tag{98}$$

where, q_i^o and q_i^f (i=1,...,7) were the initial and the final values of the volumetric flow rates at the 7 heat exchanger units, and w_j $(j = 1, \ldots, 6)$ were the weight coefficients for the six thermal service loads in the system. In Equation (97), the terms $w_4 \cdot max\left(q_4^o, q_5^o\right)$ and $w_4 \cdot max\left(\tilde{q}_4^f, \tilde{q}_5^f\right)$ reflected the fact that the two heat exchangers HEX no.4 and HEX no.5 were a redundant pair serving a single service load.

The OCR has a scale of 0 to 1 which represents a measure of how well the recovered system maintained its chilled-water delivery capacity from the level of the system before a damage. Given the formulation of Equation (97), the OCR estimation strongly relies on the right choice of the weight coefficients which represent the service load priorities based on customer requirements, mission profile, and design philosophy. In this analysis they were chosen by the author for demonstration purposes and are given in Table 16.

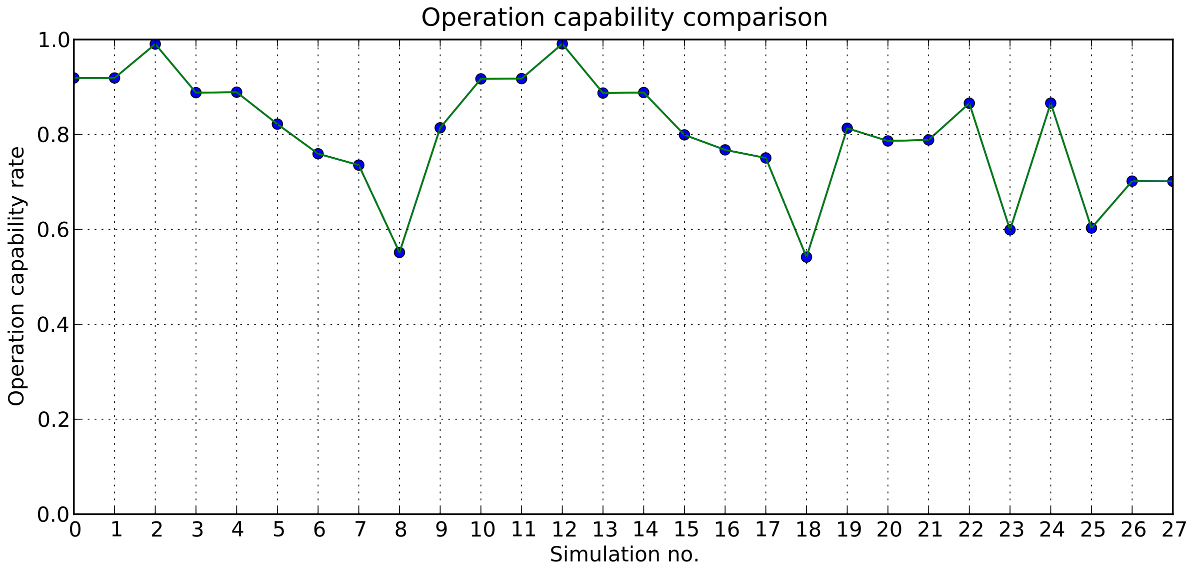
Table 16: Thermal Service Loads

Load No. :	1	2	3	4	5	6
Load Name :	IED	Eng Rm	Radar	CIC	Fuel Cell	EM Gun
w_j :	2	6	5	8	4	6
HEX No. :	1	2	3	4 5	6	7

A system's recovery performance should not be measured exclusively using the final system state after recovery, but also by how quickly the system recoveres. The OCR in Equation (97) was not formulated using this criterion because the model has no control-induced delay or failure in damage control efforts. However, in a real application to control development and test, the recovery speed must be taken into account in the formulation of OCR.

5.3.2.2 Results

Figure 76 is the OCR result of the damage analysis performed with the 28 damage cases. In Figure 76, the average value of the OCR in the damage analysis was 0.80, meaning that the system's overall capability to recover from a single rupture was quite good, although the system still had room for improvements. The result has a few interesting patterns, one of which is that simulations 8, 18, 23, and 25 not only yielded the four lowest OCR values, but also all had the same damage area in the mid-area of the port side of the system, as shown in Figure 77. This particular pattern in the analysis result clearly shows the problem area for more survivable and resilient system design. Based on the result, a design engineer can create a number of design alternatives with different schemes of control strategy, valve placement, bypass pipeline placement, or service load locations, and then perform the same routine of modeling and damage analyses introduced here to evaluate the group of design alternatives in an automated manner. Among those various analysis approaches, the analysis for optimal valve placement was performed as the next step, based on



(a) Operation Capability Rates

		Rupture location																											
		Starboard, lower					Port, lower					Starboard, upper					Port, upper					Aft		Mid				Bow	
Simulation no.		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Weight (1~10)	Compo-nents	(0, 0, 0)	(1.75, 0, 0)	(3.5, 0, 0)	(5.25, 0, 0)	(7, 0, 0)	(0, 4, 0)	(1.75, 4, 0)	(3.5, 4, 0)	(5.25, 4, 0)	(7, 4, 0)	(0, 0, 2)	(1.75, 0, 2)	(3.5, 0, 2)	(5.25, 0, 2)	(7, 0, 2)	(0, 4, 2)	(1.75, 4, 2)	(3.5, 4, 2)	(5.25, 4, 2)	(7, 4, 2)	(0, 2, 0)	(0, 2, 2)	(4, 1, 0)	(4, 3, 0)	(4, 1,2)	(4, 3, 2)	(7, 2, 0)	(7, 2, 2)
	PC # 1	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
	PC # 2	⬛	🟥	⬛	⬛	⬛	🟩	🟩	⬛	⬛	⬛	⬛	🟥	⬛	⬛	⬛	🟩	🟩	⬛	⬛	⬛	🟩	🟩	⬛	⬛	⬛	⬛	⬛	⬛
2	HEX # 1	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
6	HEX # 2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟩
5	HEX # 3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥	🟥	🟩	🟩
8	HEX # 4	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟩
	HEX # 5	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟩
4	HEX # 6	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
6	HEX # 7	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
	OCR	0.92	0.92	0.99	0.89	0.89	0.82	0.76	0.74	0.55	0.81	0.92	0.92	0.99	0.89	0.89	0.80	0.77	0.75	0.54	0.81	0.79	0.79	0.87	0.60	0.87	0.60	0.70	0.70

Com-putation Time	max	6.547 sec
	min	5.390 sec
	mean	5.073 sec

🟩 : Operating
🟥 : Down
⬛ : Turned-off

(b) System Failure Status

Figure 76: Result of Damage Analysis

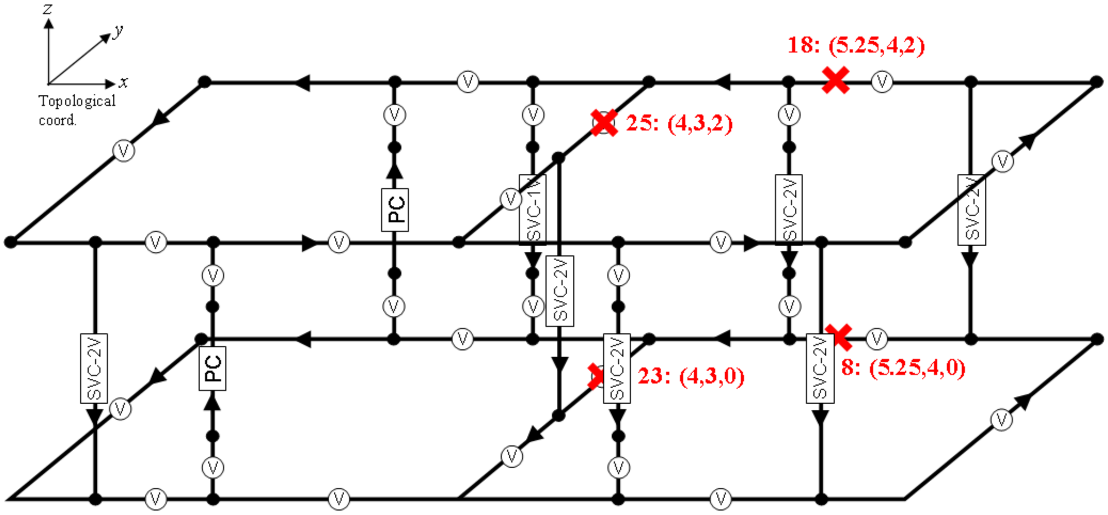


Figure 77: Rupture Locations for Damage Simulations with the Four Lowest OCR Values

the combinatorial generation of the designs with different valve placements and the evaluation of the recovery performances of them.

5.3.3 Design Analysis for Optimal Smart-Valve Placement

In the damage analysis result shown in Figure 76, another useful pattern that can be found is that the system's OCR values evaluated for simulations 0 to 9 are almost identical with the OCR values from simulations 10 to 19. This pattern can be explained by that the system's upper and lower pipelines are the exact mirror images of each other, including the valve locations in the pipelines. The rest of the simulation results conform this interpretation. Simulations 21, 24, 25, and 27, which had damages on the upper pipeline, yielded almost the same OCR values as simulations 20, 22, 23, and 26, which had damages on the lower pipeline.

Based on the above observation, the configuration of the YP-fluid system can be represented simply by the partial topology of either the upper or the lower pipeline. Similarly, the damage analysis will be sufficient with either the upper or the lower

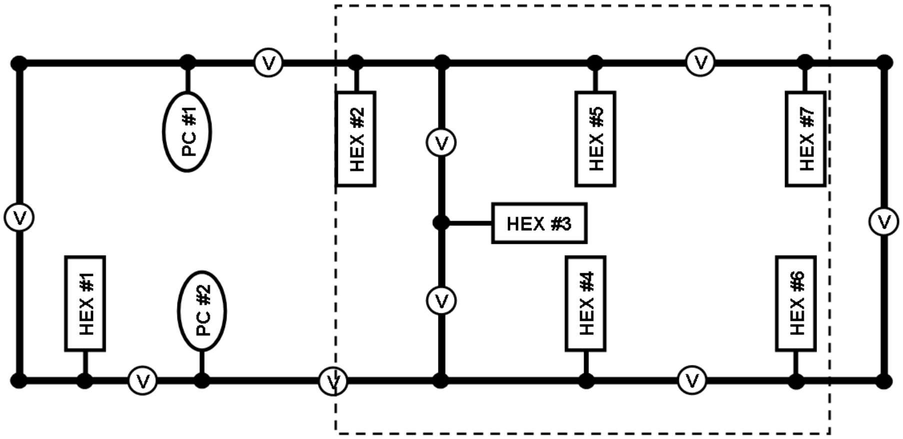


Figure 78: Original Smart-Valve Placement of YP-Fluid Model

pipeline area, because these two areas only yield redundant OCR measurement results. When the upper pipeline is chosen as the representation of the YP-fluid system, the system topology can be described by Figure 78, assuming the configuration of the lower pipeline is the exact mirror image of that of the upper pipeline. The circles with "V" indicate the smart valves for damage control.

For the simplicity and efficiency of the analysis, the focused area, which is indicated by the dotted box in Figure 78, is defined. This focused area encloses the problem area identified from the damage analysis in §5.3.2.2, which is the mid-section of the port side of the system. The analysis is not performed for the outside of the dotted box because the system already achieved considerably good recovery capability for the damage on that area.

5.3.3.1 Analysis Approach

The analysis process starts with the construction of a "bare" model by eliminating all the valves inside the dotted box from the original YP-fluid model shown in Figure 78. In the numerical implementation, the bare model is created simply by replacing all the VPIPE edge components in the dotted box into PIPE edge components. The bare YP-fluid model is shown in Figure 79. In the initialization input file for the bare model in Appendix B.2.1, *pipe* component model is assigned to all the edges in the dotted box.

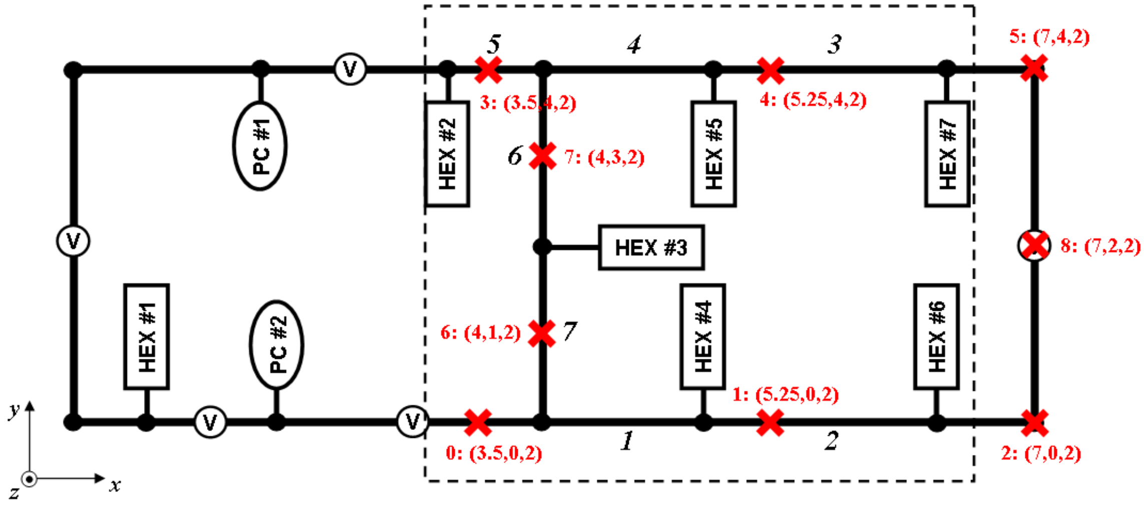


Figure 79: Bare Model of YP-Fluid System and Its Damage Locations

The bare model has seven candidate locations, which are denoted by 1 to 7 in Figure 79, for placing smart valves. The models for evaluation are generated based on the full combinatorial variations of valve placement on these seven locations, resulting in creation of the system models with 2 to 6 valves in the dotted box in Figure 79. The numbers of the generated models for the systems with different valve amounts are given in Table 17.

Table 17: Number of Generated Models for YP-Fluid Systems with Different Valve Amounts

Valve amount:	2	3	4	5	6
No. of models:	21	35	35	21	7
Total:	119				

Figure 79 also shows the damage locations for 9 simulation cases. These locations are selected from the 28 damage locations that were used in the damage analysis in §5.3.2.

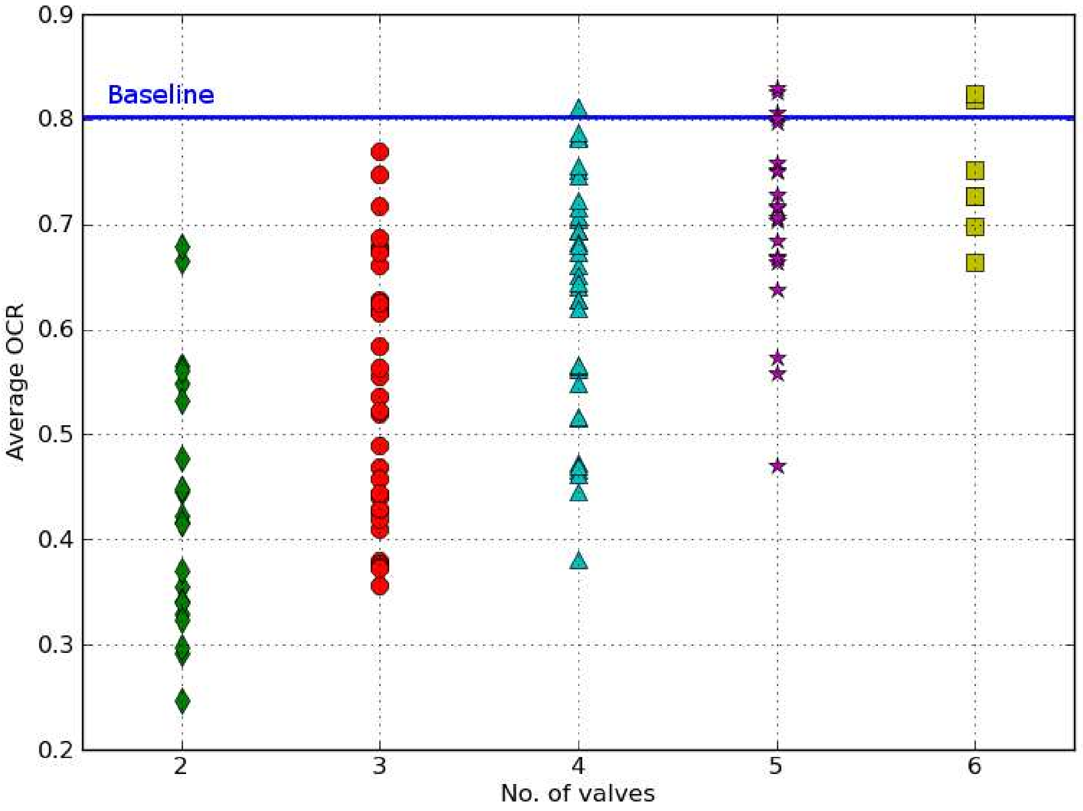


Figure 80: Number of Smart Valves vs. Average OCR

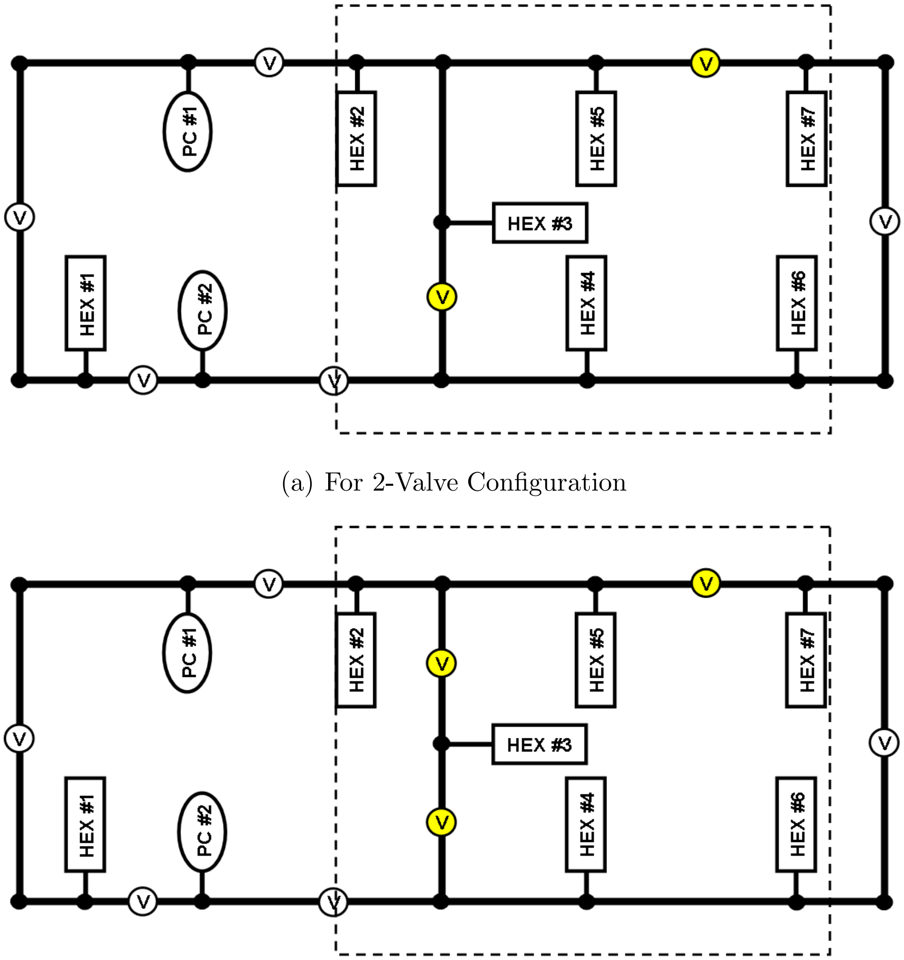
With the 9 damage locations shown in Figure 79, the same damage analysis routine as §5.3.2 is executed for each model of 119 models, providing the estimation of the OCR values of all 9 damage cases and the average OCR of each system model.

5.3.3.2 Results

Figure 80 is the plot of the average OCR values from the simulations of all 119 models. The average OCRs are plotted with respect to the number of valves that the system models have on the 7 predefined locations in the bare model, as shown in Figure 79. The blue line indicates the average OCR of the original YP-fluid model in Figure 78, which is referred to as the baseline model in the result.

According to the result of the average OCRs, there are only six system configurations that provide improvement in the overall recovery performance from the original configuration. Also comparing the results of the 5- and the 6-valve configurations, it is shown that adding one more valve from the system with 5 valves brings little or no improvement in the maximum average OCR.

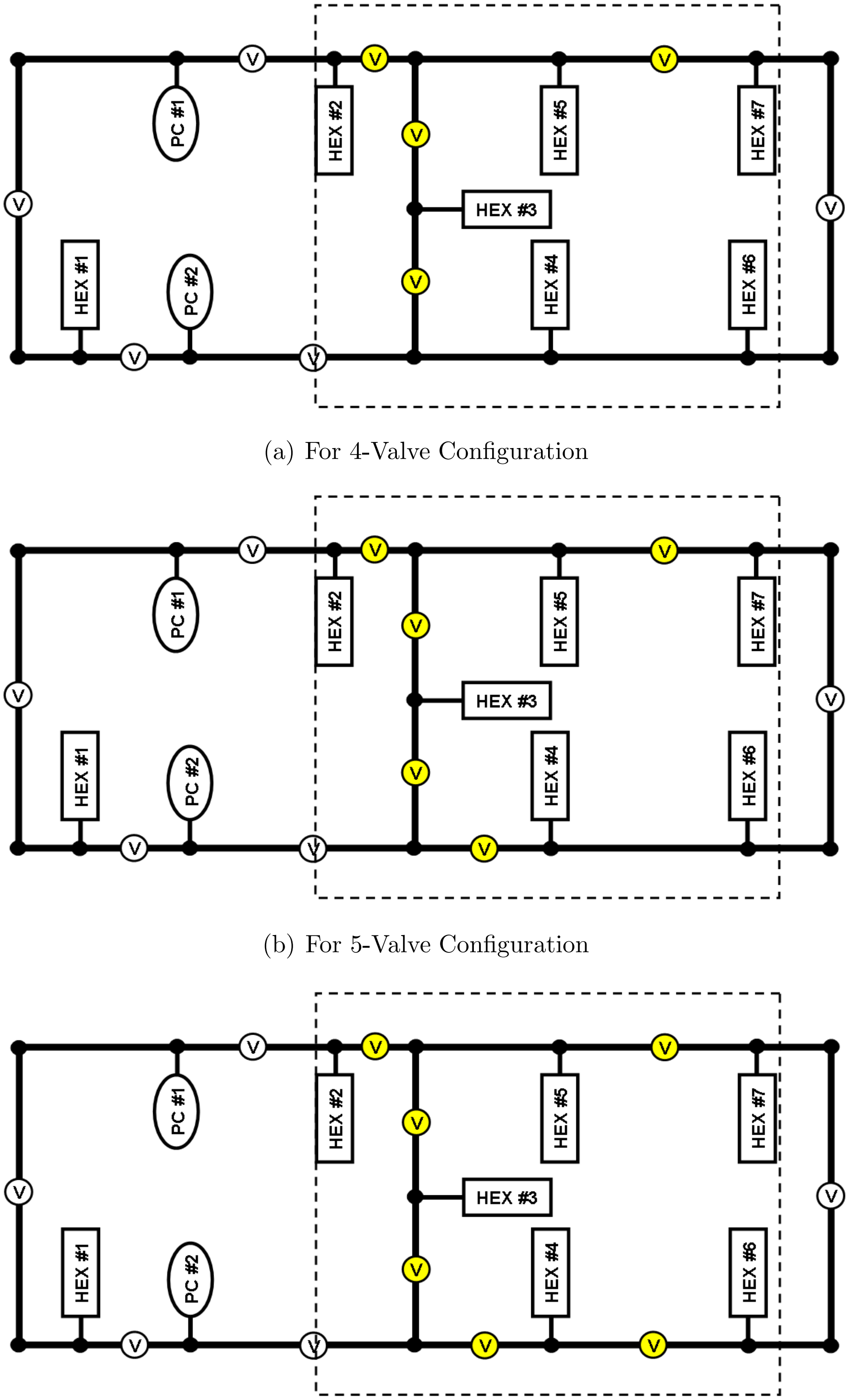
Anyway, since the goal of this analysis is to find the valve placement that provides the best recovery performance, the model with the maximum average OCR was down-selected from each group of models with the same number of valves. Figures 81 and 82 show the five models that give the maximum average OCRs among the 2- to 6-valve systems.



(a) For 2-Valve Configuration

(b) For 3-Valve Configuration

Figure 81: Smart-Valve Placement of Design with Maximum Average OCR for Different Valve Amount, Part 1



(a) For 4-Valve Configuration

(b) For 5-Valve Configuration

(c) For 6-Valve Configuration

Figure 82: Smart-Valve Placement of Design with Maximum Average OCR for Different Valve Amount, Part 2

One interesting pattern in the model topologies of the five models shown in Figures 81 and 82 is that everyone of the five models keeps the valve placement of the models with a smaller number of valves. If this pattern is true for many similar applications, knowing this pattern will be very useful for the design process for optimal valve placement, since the optimal location of every additional valve from the current design can easily be found.

Another result set is the plots of the OCR values for the 9 damage simulation cases with the five models, with comparison to the baseline model. Figure 83 shows the results of the 2-valve and the 3-valve models.

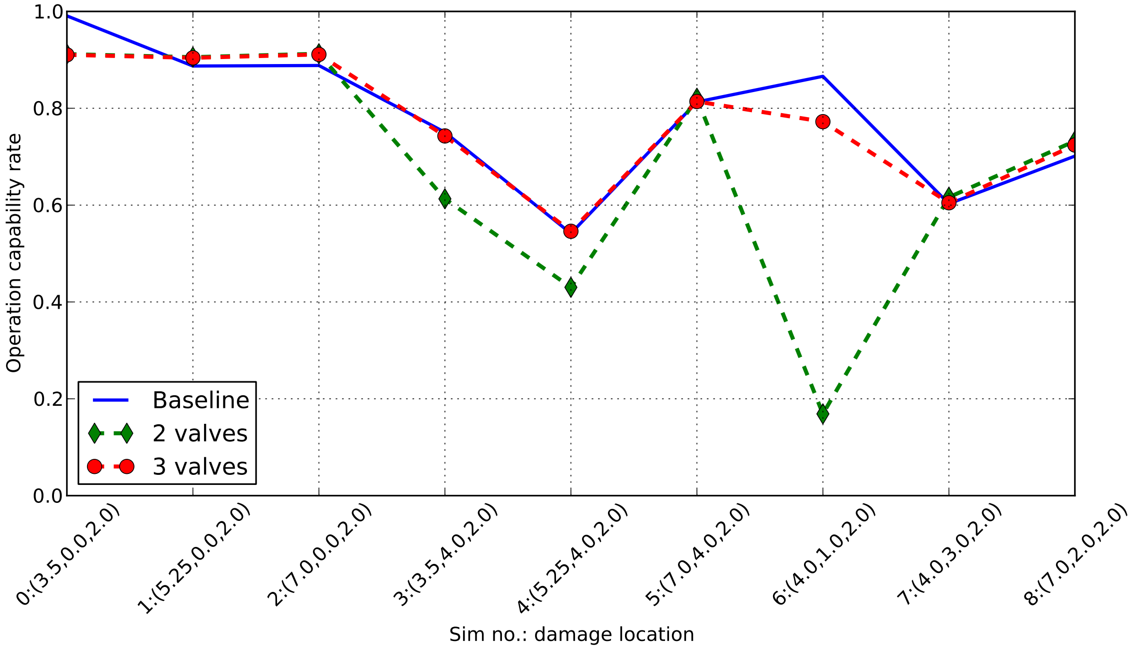


Figure 83: OCR Values of Tested Damage Cases for the Best Designs in 2- to 3-Valve Configurations

In fact, the baseline model is also one of the 4-valve models so Figure 83 shows how much degradation of the damage recovery performance is expected by decreasing the amount of smart valves from the baseline design. According to Figure 83, the design with 2-valve configuration for the focused area results in a significant level of degradation in system survivability. On the other hand, the 3-valve design yields only a subtle degradation from the baseline design.

Figure 84 shows the comparison between the baseline design, which is one of the 4-valve system, and the optimal design in the 4-valve configuration. Even with the same amount of smart valves, the optimal design delivers a more favorable OCR profile, which is flatter and more consistent through all the damage cases than that of the baseline design. Especially, the improvement on the problem area, the mid-section of the port side, is significant; for simulations 4 and 7 in Figure 84, the OCR values of the baseline model are 0.54 and 0.60, but those of the optimal model are 0.78 and 0.79.

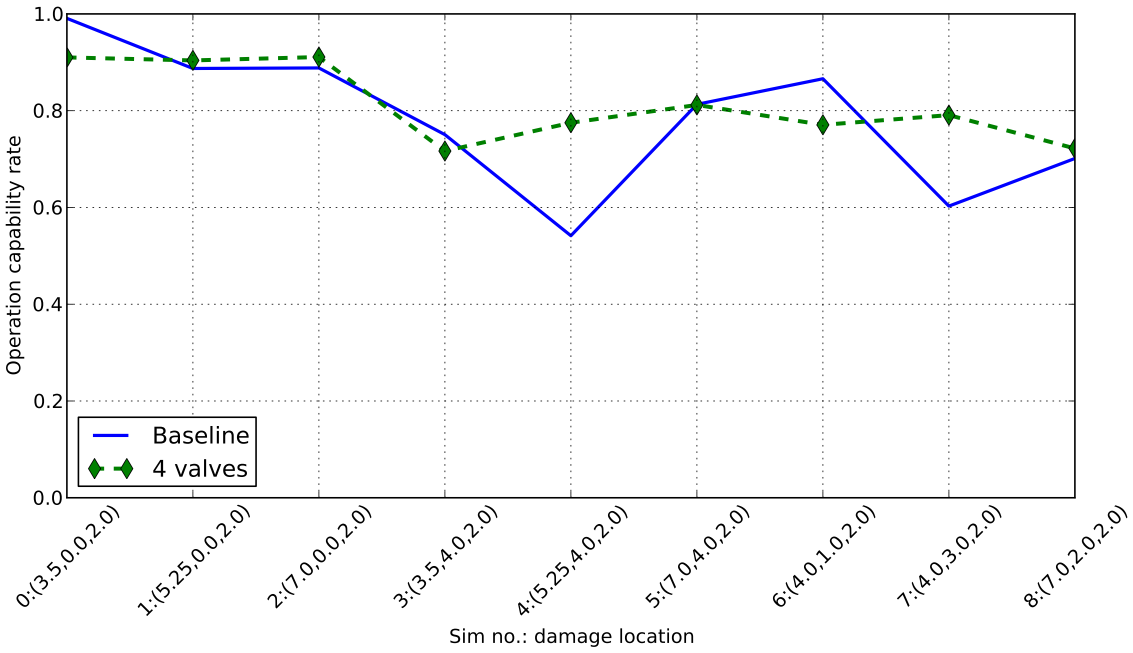


Figure 84: OCR Values of Tested Damage Cases for the Best Design in 4-Valve Configuration

Lastly, Figure 85 shows how much improvement the design with more smart valves can provide compared to the baseline and the 4-valve optimal designs. As shown in Figure 85, the 5-valve optimal design yields further improvements from the 4-valve optimal design. However, as mentioned earlier, the 6-valve design provides almost no improvement from the 5-valve design for the given damage cases. Assuming the

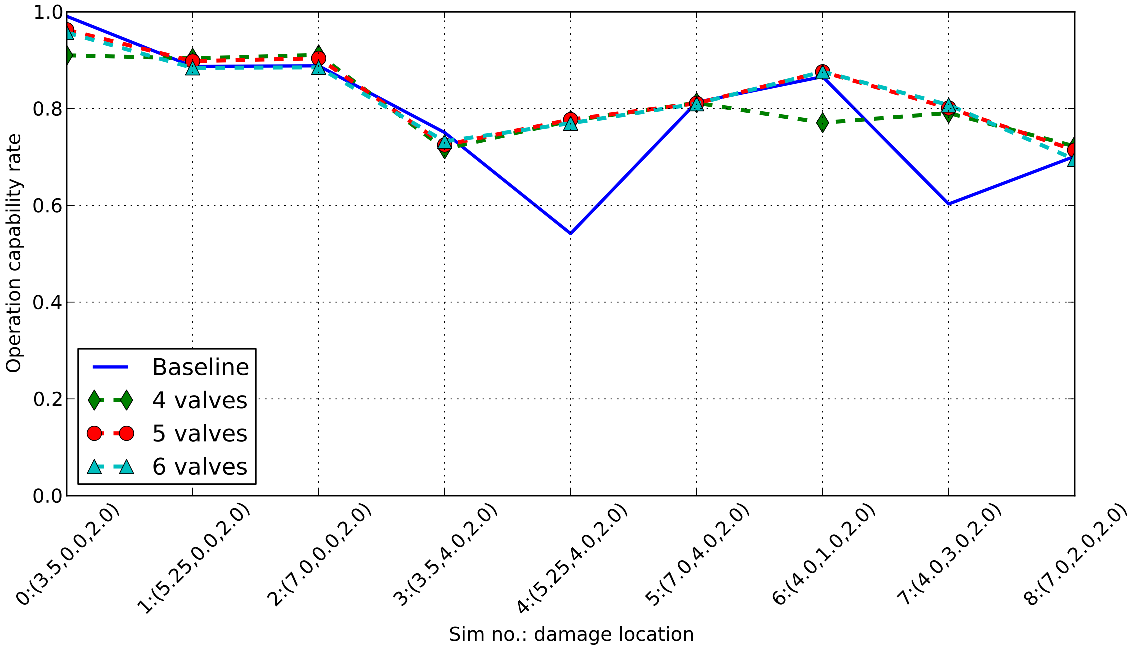


Figure 85: OCR Values of Tested Damage Cases for the Best Designs in 4- to 6-Valve Configurations

5-valve design is chosen as the final design, the final status of the system components after damages are shown for the baseline and the 5-valve designs in Figure 86, and Figure 86(b) shows that the reason for the improved OCR of the 5-valve design over the baseline is due to the improved survivability of the HEX no.2 component. In the 5-valve design, the HEX no.2 component is not down by the damage at the mid-section of the port side of the system.

5.4 Conclusions

This M&S example demonstrated a design oriented application of the developed M&S environment. First, a model of the fluid system in the notional YP was created based on the graph-based, component surrogate modeling approach, and the model's accuracy and other performances were evaluated. As a next step, a simulation-based damage analysis was performed to explore the design alternatives with various smart valve amounts and locations, as a way to find a YP-Fluid system design that could provide better survivability and resiliency to system damages or failures than the original baseline design.

The demonstration shows that the developed method is capable of delivering a computationally efficient, flexible, and automation-friendly M&S environment which can enable a more rigorous damage analysis in the early design stage. The M&S method is also expected to facilitate the design-space exploration for numerous topological and component-wise configurations, with its flexible environment that is suitable for building analysis automations.

		Rupture location								
Simulation no.		0	1	2	3	4	5	6	7	8
Weight (1~10)	Compo-nents	(3.5, 0, 2)	(5.25, 0, 2)	(7, 0, 2)	(3.5, 4, 2)	(5.25, 4, 2)	(7, 4, 2)	(4, 1,2)	(4, 3, 2)	(7, 2, 2)
	PC # 1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	PC # 2	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛
2	HEX # 1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
6	HEX # 2	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟥	🟩
5	HEX # 3	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩
8	HEX # 4	🟥	🟥	🟩	🟩	🟩	🟩	🟥	🟩	🟩
	HEX # 5	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟥	🟩
4	HEX # 6	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟥
6	HEX # 7	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟥
	OCR	0.99	0.89	0.89	0.75	0.54	0.81	0.87	0.60	0.70

🟩 : Operating
🟥 : Down
⬛ : Turned-off

(a) Baseline Model

		Rupture location								
Simulation no.		0	1	2	3	4	5	6	7	8
Weight (1~10)	Compo-nents	(3.5, 0, 2)	(5.25, 0, 2)	(7, 0, 2)	(3.5, 4, 2)	(5.25, 4, 2)	(7, 4, 2)	(4, 1,2)	(4, 3, 2)	(7, 2, 2)
	PC # 1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	PC # 2	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬛
2	HEX # 1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
6	HEX # 2	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩
5	HEX # 3	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩
8	HEX # 4	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	HEX # 5	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟥	🟩
4	HEX # 6	🟩	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟥
6	HEX # 7	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟩	🟥
	OCR	0.96	0.90	0.90	0.72	0.78	0.81	0.88	0.80	0.71

🟩 : Operating
🟥 : Down
⬛ : Turned-off

(b) 5-Valve Optimal Model

Figure 86: Component Status Comparison Between Baseline and 5-Valve Optimal Designs of YP-Fluid System

CHAPTER VI

CONCLUSIONS

In this chapter, the approaches and developments in this thesis are reviewed based on the research problems identified in §1.3. Then, the drawbacks and limitations of the developed modeling method are pointed out, and the directions of the future research for improving the modeling method are also discussed from those weaknesses and limitations. Lastly, the future research directions in the view of the potential extension of the application domain of the thesis work are briefly introduced.

6.1 Review Based on Research Problems and Goals

The research motivation of the thesis started from the observation of the problems in the current simulation-based approaches for developing fail-proof, integrated engineering systems of a naval military platform. Among many problems, the author specifically focused on the capability gaps in the domain-level simulation for the simulation-based design of fail-proof engineering systems. These identified gaps were damage modeling capability, and flexible model reconfiguration, and simulation speed as found in §1.3. In order to answer the solutions to the three problems, the author developed the graph-based component surrogate modeling method, and the modeling method is reviewed in order to check how well it addresses the three problems to be solved.

6.1.1 Problem 1: Damage Modeling

The fundamental elements of the solution approach, not only for the problem of damage modeling, but also for the other two problems, are the graph-based topological modeling method and the separated management of the component behavior model library. The two elements in the M&S formulation are the key enablers to build highly flexible fluid network models that are reconfigurable both before and during simulation, but these two elements are not sufficient for modeling damages of fluid systems. As the application-specific solution for augmenting the damage modeling feature on top of the basic two elements for flexible and reconfigurable modeling environment, the damage modeling tools such as damage bubble class, reference control model, and the auto-generation algorithm of reference control model are developed. As a result, as demonstrated in §5.3.2, the model based on the developed M&S approach exhibited a high level of rigorousness in damage analysis that could not be expected from currently available pipeline fluid-modeling tools.

6.1.2 Problem 2: Model Reconfiguration

Along with the graph topological modeling and the component model library, the object-oriented script-based modeling environment contributes the environment for automated model reconfiguration. There can be three different cases in model reconfiguration, which are, changes of connection topology without changes in behavior models of the components, changes of behavior models of components without changing their connectivity, and changes of both component behaviors and their topologies. The first case is easily programmable in a way to changing the component models of edges to any other models in the component library, which is managed by linking a selected model function to the *model* attribute of edge objects. For the second case, the incidence matrix of graph theory as the instantiation of the connection topology of the fluid model makes the connection topology of the model highly reconfigurable both before and during the simulation. The third case can be implemented by combining the two modeling features for the first two cases. Among the three cases, the first case was demonstrated in §5.3.3, and for the second and third cases, damage modeling and simulation is a specific use-case of the third case, so they were indirectly demonstrated in §5.3.2 and §5.3.3.

6.1.3 Problem 3: Simulation Cost

In order to improve the speed of the simulation-based analysis for the naval fluid system, the M&S method applies the surrogate modeling technique for generating dynamic component behavioral models. In §3.1, the requirements were set for the surrogate modeling approach applied to the fluid system components, and they are summarized as follows:

1. Sufficient level of fidelity in nonlinear dynamic systems modeling.
2. Model parsimony for computational efficiency.
3. Surrogate modeling process that is simple, efficient, and automation-friendly, with minimal data generation required.

Based on the literature search of available surrogate modeling techniques and methods in Chapter 2, RNN was selected as the most suitable surrogate model structure among them, but the surrogate modeling approach based on a plain RNN still needed improvements in model stability and robustness, and training efficiency to satisfy all the requirements. As a result, the output-feedback block structure for RNN and various other tools were developed for RNN-based surrogate modeling.

The RNN-based surrogate modeling method was validated in the two example implementations in §3.4, showing the significant improvements in not only model stability and robustness but also model training efficiency and accuracy over the plain RNN. The improvements of model stability and accuracy also indirectly mean the reduced data size required for training RNN. In order to ensure the model stability and accuracy of a plain RNN, the training data must fill the response space very densely, and the trained RNN should be validated with the separate validation data set. However, the block structure in §3.2 augments the RNN's model stability so significantly that the RNN with this block structure does not have to rely on the massive size of a training data set or a validation data set in order to ensure model stability and robustness.

6.2 Drawbacks and Limitations

The developed modeling method has weaknesses and limitations, which provide open questions for future research. In this section, some of the weaknesses and limitations that were identified by the author are briefly introduced and discussed.

6.2.1 Limited Data Accessibility of Component Surrogate Models

The developed modeling method applies the RNN-based surrogate modeling approach in order to create component models of a fluid network system. The drawback of the surrogate modeling approach is that, once the surrogate model is created, it is impossible to access any data or information other than the model outputs defined during the surrogate model generation process. In order to access the additional data, the surrogate model should be regenerated with a new training data set which is generated by the fresh execution of computer experiment. For such components, therefore, physics-based modeling will be a better choice than surrogate modeling.

6.2.2 Difficulties in Predicting and Controlling Accuracy of Component Surrogate Models

A big problem of system modeling based on the aggregation of component surrogate models is that the model accuracy is extremely difficult to predict. This error-prediction problem can be summarized to the two smaller problems of:

1. Proper definition of model error for dynamic systems.
2. Identification of the mechanism of error propagation in an aggregated model.

The difficulty in answering the first question was from the feedback structure of the dynamic simulation model. For a static model, the model accuracy can be measured by the error between the actual and the predicted outputs. For a dynamic model, the predicted output is fed back to the model in order for the model to generate the output for the next time step, meaning that the error in the predicted output is also fed into the model and affects the model output for the next time step. This error propagation, or the error dynamics, of the dynamic model is very important because the instability of the error dynamics is the main cause of the model instability. In the implementation examples in §3.4 and §5.3, the MSEs measured from the training and test data sets were used as the measurement of model accuracy of all the surrogate models, but these two MSEs are not the adequate choice of the model accuracy measurement for the dynamic models since they are lack of the information about the stability of error dynamics.

The second problem is about knowing how the error is propagated from one component model to another. This mechanism of error propagation in the system is believed to be dependent on both the connection topology of its component models and the local error dynamics of each component model, which again, leads to the first problem. Because of the complexity in predicting the error for the type of system models covered in the thesis, it is mostly left as the part of future research.

6.2.3 Linear Edge-Based Component Definition

The developed modeling method only allows the linear edge, which comprises two nodes and a single flow property as the definition of components. This single way of defining components can be troublesome when it comes to modeling with components that consist of multiple nodes and flows and can not be decomposed any further into linear edge components. A good example of such components in fluid systems is a T-junction, which has three nodes and three different flow values.

In the graph-based topological modeling method, there is another limitation, which is, a system with multi-graph topology can not be implemented. The current numerical implementation of the connection topology is heavily utilizing the incidence matrix of linear graph theory, which does not have capability of modeling several characteristic elements of multi-graph such as self-loops and multiple edges on common two nodes.

6.2.4 Modeling of Compressible-Flow Systems

A component in the developed modeling method is defined by an edge of a graph, and the flow through an edge is assumed to be unique. However, for a compressible-flow system, this assumption is not true. In order for the graph-based component-surrogate modeling method to be applicable to compressible-flow systems, its component definition and component modeling process must be further developed, or modified, to address the flow value changes within edges caused by the compressibility effect.

6.3 Future Research

Although the graph-based component surrogate modeling method has been developed based upon the applications to a ship fluid system, it has potential merits as the solutions of many M&S challenges in different domains of application.

6.3.1 Application Expansion to Electric Power Distribution Systems

The DC electric power networks have strong physical analogies to fluid network systems so it may be a natural path of the future research to expand the application of the graph-based component surrogate modeling method to electrical systems modeling. This expansion of the application domain can bring synergistic benefits in simulation-based design of many large-scale engineering systems, since the today's "more-electric" systems have strong coupling of electric power and thermal management using fluid systems, and the graph-based M&S environment for the integrated analysis of both the electrical and the fluid cooling systems will be an enabler to bring a more sophisticated and holistic trade study of a large-scale engineering system into early design stage.

6.3.2 Simulation-Based Analysis for Energy Optimized Aircraft (EOA)

One of the goals in Integrated Vehicle & Technology (INVENT) program initiated by the U.S. Air Force Research Laboratory (AFRL) is to bring the platform-level energy optimization early into the design process of today's more electric, more integrated aircraft [82]. This approach has led to a multitude of challenges, and one of them is that the aircraft is composed of a large number of components that are modeled using physics-based, time-domain modeling, so the resulting high-fidelity aircraft model becomes computationally too expensive to use in the mission-level or system-level optimization analyses. In order to address this problem, AFRL tries to perform system-level M&S by generating the reduced-fidelity models of various subsystems and components, such as thermal management, electric power distribution and management, and electronics devices.

Considering the AFRL's M&S approach, the RNN-based surrogate modeling method developed in this thesis can be a solution for fast and robust creation of the reduced-order time-domain models for the components such as fluid-based cooling units, electric power components, and electronics devices in the aircraft system. In addition to the RNN-based surrogate modeling approach, the graph-based topological modeling method for weaving the components for fluid and electrical components can provide a flexible and dynamic modeling environment for system-level optimization in various aspects including energy, sensor, and survivability optimizations.

APPENDIX A

FLOW CONTINUITY-CHECK THRESHOLD OF SMART-VALVE CONTROL AGENTS

As previously described in §4.2.2 and §4.2.2.1, the reference damage control system is an aggregated system of control units embedded on the system components. These component-level control units are attached at smart valves and pump-chiller subnetworks, and close their valves if flow discontinuities are detected by comparing the flow rates measured from the neighboring controllers with those measured from their own valves using the criterion in Equation (85). This criterion uses ε as a threshold for determining whether flow is continuous or not, which means that different selection of the value of ε can affect the control unit's performance of detecting a rupture around it, and furthermore, the performance of the reference damage control model.

This appendix provides a short case-study of testing the effect of the flow continuity threshold value *varepsilon* to the rupture isolation performance of the reference damage control model. As the test platform, the notional YP-fluid model in Chapter 5 and its reference damage control model are reused.

Logically, the lower bound of the possible *varepsilon* value is determined by the numerical tolerance used in the iterative solution process in the simulation, which is Algorithm 2 in §4.3.2. If ε in the control units is set smaller than this numerical tolerance, the reference damage control model will not be able to distinguish between the event of flow discontinuity and the numerical error of the simulation, raising the false detection of system rupture. The numerical tolerance of the iterative solution process in the simulation of the YP-fluid model is set to 10^{-7}. This is an absolute tolerance applied for checking the numerical convergence of the norm of vector h in

Algorithm 2, which is the vector of flow sums at the nodes, and represents the KCL constraints of the graph-based model. In the simulation of the YP-fluid model, there is a secondary tolerance that is set to 8×10^{-6}. Since the component behavioral models for the edges of the system are all RNN surrogate models, the simulation sometimes have difficulties in being converged within the accuracy of the original convergence tolerance when the simulation happens to be performed at the operation region that is outside the input spaces of some of the surrogate models. In that case, the secondary tolerance works as a relaxed convergence criterion. If this secondary tolerance is also violated in the simulation, then the simulation process sends a warning message, and does not feed the model response output to the control system model in order to prevent the control model from falling into false detection of ruptures. Thus, with considering 10^{-7} as a hard limit, and 8×10^{-6} as a soft limit, the flow continuity threshold ε of the control units for the YP-fluid system model was set to 1.5×10^{-5} in the demonstration of Chapter 5.

The test is performed by executing the same damage experiment as the one with 28 damage cases in §5.3.2, with five different values of ε, which are 10^{-6}, 1.5×10^{-5}, 10^{-4}, 5×10^{-4}, and 10^{-3}. The first value is between the hard and soft lower limits of ε, and the second value is the same as the one used in the demonstration of Chapter 5. As in §5.3.2, the result of each damage experiment is presented by the plot of OCR vs. simulation numbers, which is Figure 87. In Figure 87, all the damage experiments yield almost the identical OCR results, except for the one with the largest setting of ε, which is 10^{-3}. Actually, the results show that the control system with $\varepsilon = 10^{-3}$ fails to detect and isolate ruptures in many of the 28 different damage cases. Figure 88 reveals the reason for the control system's failure to detect and isolate ruptures. Figure 88 is the plot of the total rupture flow rates result from simulation no.9 in all five damage experiments with different ε values. In simulation no.9, a rupture is given at (7, 4, 0), as shown in Figure 74. Figure 88 shows that the control reaction to the rupture

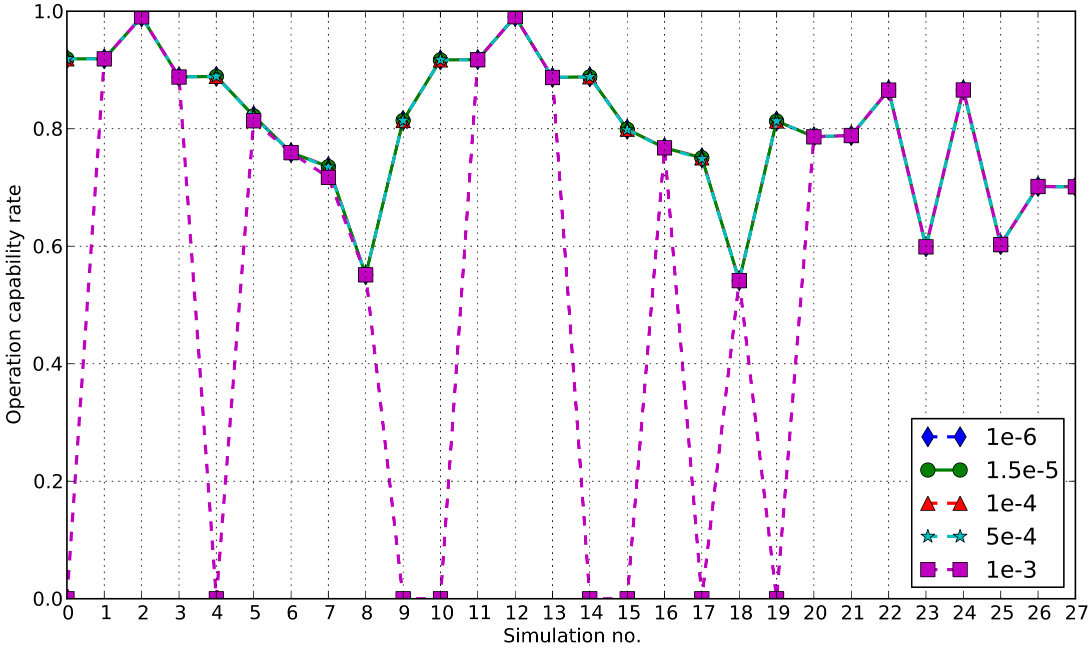


Figure 87: Damage Analysis Results with Different Values of ε in Control Units

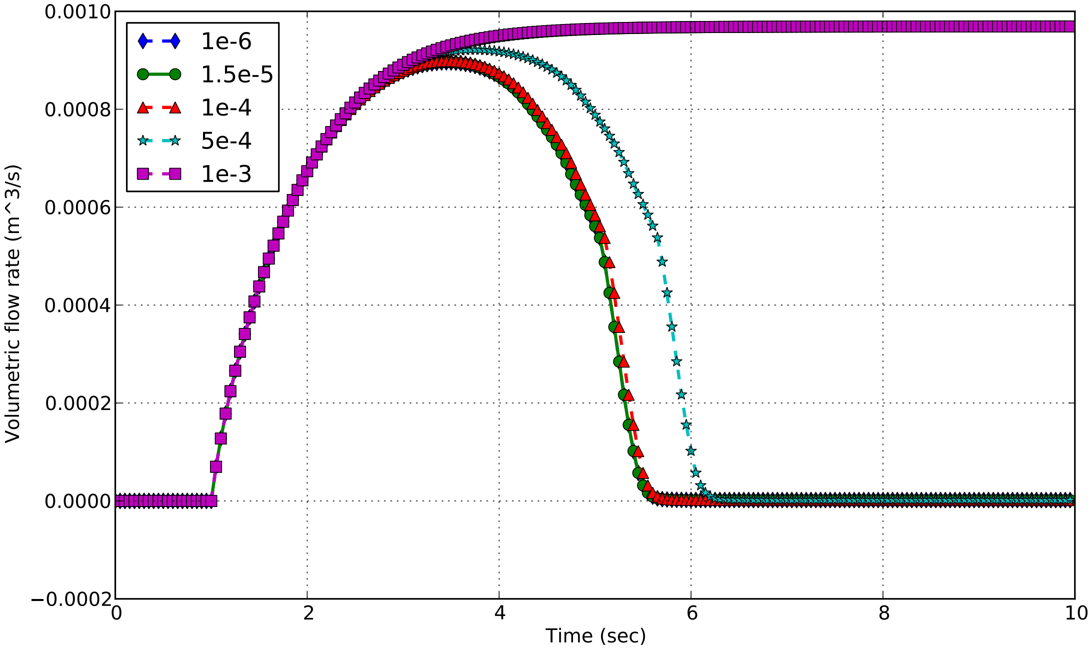


Figure 88: Total Rupture Flow Rate from Simulation No.9 with Different Values of ε in Control Units

becomes slightly delayed as the value of ε becomes larger. With $\varepsilon = 10^{-3}$, the control system can not detect the rupture any more, since the magnitude of the rupture flow in the system is still smaller than the threshold $\varepsilon = 10^{-3}$ in the control units. The result clearly shows that the selection of ε for control units must be bounded by the expected magnitude of the rupture flow rate in the damage analysis, in order to develop a reliable reference damage control model.

APPENDIX B

SOURCE CODE

The chapter provides Python execution mains, input script files, codes of model libraries created for the demonstrations in Chapter 5. Table 18 is the list of the Python module files, with the brief description of them. The M&S environment implemented in Python consists of three parts of program codes, which are a model initialization script file, an execution-main script, and the *my_modules* package. The *my_modules*

Table 18: Python Files for M&S of Notional YP-Fluid System

Package/Module-File		Description
	notional_yp.txt (B.1.1):	Graph-based model initialization file input for generating the graph-based fluid system model in §5.3.2. Has the configuration information of the entire fluid network model.
	yp_run3.py (B.1.2):	Analysis execution main for the demonstration in §5.3.2.
	yp_bare_model.txt (B.2.1) :	Initialization file input for generation the bare model in §5.3.3.
	comb_anal.py (B.2.2):	Analysis execution main for the demonstration in §5.3.3.
my_modules	cen_model.py (B.3.1):	Contains node, edge, and graph classes.
	components.py (B.3.2):	Contains all component RNN-surrogate models
	simul.py (B.3.3):	Contains damage planner, damage bubble, and solver classes.
	control.py (B.3.4):	Contains component-level control agent, and reference damage control system classes.
	post_proc.py (B.3.5):	Has *Recorder* class for storing, replaying, visualizing the simulation results, and a few other function for post-processing and analyzing the results.

package is a package of the Python modules (files containing the definitions of files and classes) that are the implementation of the methods and tools developed in this thesis. For details regarding the codes, refer to the comments in them.

B.1 Execution Main and Initialization Scripts for for §*5.3.2*

B.1.1 notional_yp.txt

The first three lines of the initialization script are the definition of the lower and upper bounds for three dimensional topological coordinates. In the two dimensional case, the lower and upper bounds for z-coordinate are all set to zero.

After the "edge:" keyword, each line defines the configuration of an edge, whose information is separated by spaces or tabs. The configuration information is given in the following order in a single line: the topological coordinates of the in-node, topological coordinates of the out-node, function name of a component behavioral model linked to the edge, type of the edge (normal, source, sink, or damaged), name of the edge component, control input variables and their initial values, and model parameters and their values.

When the defined in-node or out-node is a reference node, then "ref@" must be added in front of the coordinate. For assigning a component behavioral model to the edge, the assigned model function also must exist in the component library, which is components.py module.

```
# Lower and upper bounds of the topological coordinates.
#           Lower Upper
x_bound:    0  7
y_bound:    0  4
z_bound:    0  3
# NodeIn, NodeOut, model-func-name, edge-type, model-name,
# ctrl inputs, params
# choices of node types: nrml(default), dmg, ref
# choices of edge types: nrml, dmg, src, snk
```

```
edge:
[0,0,0] [1,0,0]     pipe    nrml    pipe_lp_1   {}          {’l’:3.048}
[1,0,0] [2,0,0]     vpipe   nrml    pipe_lp_2   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[2,0,0] [4,0,0]     vpipe   nrml    pipe_lp_3   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[4,0,0] [5,0,0]     pipe    nrml    pipe_lp_4   {}          {’l’:3.048}
[5,0,0] [6,0,0]     vpipe   nrml    pipe_lp_5   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[6,0,0] [7,0,0]     pipe    nrml    pipe_lp_6   {}          {’l’:3.048}
[7,0,0] [7,4,0]     vpipe   nrml    pipe_lp_7   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[7,4,0] [6,4,0]     pipe    nrml    pipe_lp_8   {}          {’l’:3.048}
[6,4,0] [5,4,0]     vpipe   nrml    pipe_lp_9   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[5,4,0] [4,4,0]     pipe    nrml    pipe_lp_10  {}          {’l’:3.048}
[4,4,0] [3,4,0]     pipe    nrml    pipe_lp_11  {}          {’l’:3.048}
[3,4,0] [2,4,0]     vpipe   nrml    pipe_lp_12  {’v1’:1}    {’r’:0.0254,’l’:6.096}
[2,4,0] [0,4,0]     pipe    nrml    pipe_lp_13  {}          {’l’:3.048}
[0,4,0] [0,0,0]     vpipe   nrml    pipe_lp_14  {’v1’:1}    {’r’:0.0254,’l’:6.096}
[4,4,0] [4,2,0]     vpipe   nrml    bps_lp_1    {’v1’:1}    {’r’:0.01905,’l’:6.096}
[4,2,0] [4,0,0]     vpipe   nrml    bps_lp_2    {’v1’:1}    {’r’:0.01905,’l’:6.096}
[0,0,2] [1,0,2]     pipe    nrml    pipe_hp_1   {}          {’l’:3.048}
[1,0,2] [2,0,2]     vpipe   nrml    pipe_hp_2   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[2,0,2] [4,0,2]     vpipe   nrml    pipe_hp_3   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[4,0,2] [5,0,2]     pipe    nrml    pipe_hp_4   {}          {’l’:3.048}
[5,0,2] [6,0,2]     vpipe   nrml    pipe_hp_5   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[6,0,2] [7,0,2]     pipe    nrml    pipe_hp_6   {}          {’l’:3.048}
[7,0,2] [7,4,2]     vpipe   nrml    pipe_hp_7   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[7,4,2] [6,4,2]     pipe    nrml    pipe_hp_8   {}          {’l’:3.048}
[6,4,2] [5,4,2]     vpipe   nrml    pipe_hp_9   {’v1’:1}    {’r’:0.0254,’l’:6.096}
[5,4,2] [4,4,2]     pipe    nrml    pipe_hp_10  {}          {’l’:3.048}
[4,4,2] [3,4,2]     pipe    nrml    pipe_hp_11  {}          {’l’:3.048}
[3,4,2] [2,4,2]     vpipe   nrml    pipe_hp_12  {’v1’:1}    {’r’:0.0254,’l’:6.096}
[2,4,2] [0,4,2]     pipe    nrml    pipe_hp_13  {}          {’l’:3.048}
[0,4,2] [0,0,2]     vpipe   nrml    pipe_hp_14  {’v1’:1}    {’r’:0.0254,’l’:6.096}
[4,4,2] [4,2,2]     vpipe   nrml    bps_hp_1    {’v1’:1}    {’r’:0.01905,’l’:6.096}
[4,2,2] [4,0,2]     vpipe   nrml    bps_hp_2    {’v1’:1}    {’r’:0.01905,’l’:6.096}
```

```
[2,4,1.8] [2,4,2]   vpipe   nrml    out_pc1     {'v1':1}    {'r':0.01905,'l':0.6096}
ref@[2,4,1] [2,4,1.8] cp,0,snk,0 src pc1_out  {'v1':1,'v2':0,'ps1':200,'ps2':0}   {}
[2,4,0.2] ref@[2,4,1] cp,1,src,0 snk pc1_in   {}  {}
[2,4,0] [2,4,0.2]   vpipe   nrml    in_pc1      {'v1':1}    {'r':0.01905,'l':0.6096}
[2,0,1.8] [2,0,2]   vpipe   nrml    out_pc2     {'v1':0}    {'r':0.01905,'l':0.6096}
ref@[2,0,1] [2,0,1.8] cp,0,snk,0 src pc2_out  {'v1':1,'v2':0,'ps1':0,'ps2':0}   {}
[2,0,0.2] ref@[2,0,1] cp,1,src,0 snk pc2_in   {}  {}
[2,0,0] [2,0,0.2]   vpipe   nrml    in_pc2      {'v1':0}    {'r':0.01905,'l':0.6096}
[1,0,2] [1,0,0]     svc_2v  nrml    svc1        {'v1':1,'v2':0} {}
[3,4,2] [3,4,1.8]   vpipe   nrml    in_svc2     {'v1':1}    {'r':0.0127,'l':1.2192}
[3,4,1.8] [3,4,0.2] svc_1v  nrml    svc2        {'v1':1}        {}
[3,4,0.2] [3,4,0]   vpipe   nrml    out_svc2    {'v1':1}    {'r':0.0127,'l':1.2192}
[4,2,2] [4,2,0]     svc_2v  nrml    svc3        {'v1':1,'v2':0} {}
[5,0,2] [5,0,1.8]   vpipe   nrml    in_svc4     {'v1':1}    {'r':0.0127,'l':1.2192}
[5,0,1.8] [5,0,0.2] svc_2v  nrml    svc4        {'v1':1,'v2':0} {}
[5,0,0.2] [5,0,0]   vpipe   nrml    out_svc4    {'v1':1}    {'r':0.0127,'l':1.2192}
[5,4,2] [5,4,1.8]   vpipe   nrml    in_svc5     {'v1':1}    {'r':0.0127,'l':1.2192}
[5,4,1.8] [5,4,0.2] svc_2v  nrml    svc5        {'v1':1,'v2':0} {}
[5,4,0.2] [5,4,0]   vpipe   nrml    out_svc5    {'v1':1}    {'r':0.0127,'l':1.2192}
[6,0,2] [6,0,0]     svc_2v  nrml    svc6        {'v1':1,'v2':0} {}
[6,4,2] [6,4,0]     svc_2v  nrml    svc7        {'v1':1,'v2':0} {}
```

B.1.2 yp_run3.py

```
from my_modules.cen_model import *
from my_modules.simul import *
from my_modules.control import *
import my_modules.components as comps
from my_modules.post_proc import *
from numpy import *
import cPickle
import copy
import time
```

```

# ---------------------------------------------------------------------------
# Generate a library of component model functions.

c_lib = dict()
# Svc load with a single valve:
c_lib['svc_1v']=comps.svc_1v

# Svc load with double valves:
c_lib['svc_2v']=comps.svc_2v

# Chiller-pump unit:
c_lib['cp']=comps.cp

# Pipe without valve:
c_lib['pipe']=comps.pipe

# Pipe with valve:
c_lib['vpipe']=comps.vpipe


# ---------------------------------------------------------------------------
# Creation of a graph model

comp_ts = time.clock()

# Feed the 'notional_yp.txt' file and the component libary to create
# a graph model.
mdl = GraphModel('./notional_yp.txt', c_lib)

# Initialize the node pressure.
n_nodes = mdl.A.shape[0]
```

```
ini_np = 135000*ones(n_nodes)
mdl.set_np(ini_np)


dt = 0.05 # time step of simulation.
slvr = Solver(mdl,dt,1e-7,5) # Initialize solver object.

# Run simulation to obtain initial conditions.
for i in xrange(70):
    slvr.run_to_next_time_step()
mdl.empty_data()                # Reset the data storage
mdl.t=0                         # Reset the simulation time
ini_mdl = cPickle.dumps(mdl,1)  # Store the initial model.
#---------------------------------------------------------------------
# Damage simulation.

t_f = 10     # Final time of simulation.

# Damage ranges and grid density.
x_set = [0,7,5]
y_set = [0,4,2]
z_set = [0,2,2]
d_rad = 0.7          # Damage bubble radius.
t_dmg = 1            # Time that a damage occurs.

# Create damage set.
dmg_set = DamageSet(x_set,y_set,z_set,d_rad,t_dmg)

# Additional customization of the damage set.
more_set = [[0,2,0],[0,2,2],[4,1,0],[4,3,0],\
                                        [4,1,2],[4,3,2],[7,2,0],[7,2,2]]
dmg_set.damage_set = append(dmg_set.damage_set,more_set,axis=0)
dmg_set.size = dmg_set.damage_set.shape[0]

# Setting default component model and parameters for damaged components
dflt_model = c_lib['pipe']
dflt_prm = {'l':0.5,'r':0.0254}
rupt_press = 110000          # Pressure setting for the damage nodes

# Initialize data recorder
recorder = Recorder()
comp_t = (time.clock()-comp_ts)
print "init", comp_t
time_dat = zeros(dmg_set.size)

# Damage simulation body:
for i in xrange(dmg_set.size):
    comp_ts = time.clock()
    mdl_2 = cPickle.loads(ini_mdl)    # Load the initial graph model
    slvr.set_model(mdl_2)             # Connect the model to solver

    # Create the reference damage control model
    dcs = DmgCtrlSys()
    dcs.setup_cp_agents(mdl_2,dt)
    dcs.setup_smart_valves(mdl_2,dt)
    dcs.connect(mdl_2)

    # Update controllers in the model.
    for ag in dcs.agent:
        ag.update_self()
    for ag in dcs.agent:
        ag.update_ports()

    # Load damage bubble from damage set.
    dmg_bbl = dmg_set.go_to_set(i)
    dmg_bbl.dflt_model = dflt_model
    dmg_bbl.dflt_prm = dflt_prm

    # Connect the bubble and data recorder objects to the solver.
    slvr.dmg_bubble = dmg_bbl
    slvr.recorder = recorder

    # Run damage simulation until the final time t_f.
    t = 0
    while t <= t_f:
        if slvr._conv_flag == 1:
            for ag in dcs.agent:
                ag.update_self()
            for ag in dcs.agent:
                ag.update_ports()
            for ag in dcs.agent:
                ag.process()
        slvr.run_to_next_time_step()
        t +=dt
    comp_t = (time.clock()-comp_ts)

    # Store the entire model into the data recorder object.
    recorder.stack_model(mdl_2)
    recorder.stack_res()
    print i,comp_t
    time_dat[i] = comp_t
```

B.2 Execution Main and Initialization Scripts for §5.3.3

B.2.1 yp_bare_model.txt

```
# Lower and upper bounds of the coordinates of
# node positions.
#            Lower Upper
```

```
x_bound:    0  7

y_bound:    0  4

z_bound:    0  3

# NodeIn, NodeOut, c-model-type, model-name, ctrl inputs, params

# choices of node types: nrml(default), dmg, ref

# choices of edge types: nrml, dmg, src, snk

edge:

[0,0,0] [1,0,0]     pipe    nrml    pipe_lp_1   {}          {’l’:3.048}

[1,0,0] [2,0,0]     vpipe   nrml    pipe_lp_2   {’v1’:1}    {’r’:0.0254,’l’:6.096}

[2,0,0] [4,0,0]     vpipe   nrml    pipe_lp_3   {’v1’:1}    {’r’:0.0254,’l’:6.096}

[4,0,0] [5,0,0]     pipe    nrml    pipe_lp_4   {}          {’l’:3.048}

[5,0,0] [6,0,0]     pipe    nrml    pipe_lp_5   {}          {’l’:6.096}

[6,0,0] [7,0,0]     pipe    nrml    pipe_lp_6   {}          {’l’:3.048}

[7,0,0] [7,4,0]     vpipe   nrml    pipe_lp_7   {’v1’:1}    {’r’:0.0254,’l’:6.096}

[7,4,0] [6,4,0]     pipe    nrml    pipe_lp_8   {}          {’l’:3.048}

[6,4,0] [5,4,0]     pipe    nrml    pipe_lp_9   {}          {’l’:6.096}

[5,4,0] [4,4,0]     pipe    nrml    pipe_lp_10  {}          {’l’:3.048}

[4,4,0] [3,4,0]     pipe    nrml    pipe_lp_11  {}          {’l’:3.048}

[3,4,0] [2,4,0]     vpipe   nrml    pipe_lp_12  {’v1’:1}    {’r’:0.0254,’l’:6.096}

[2,4,0] [0,4,0]     pipe    nrml    pipe_lp_13  {}          {’l’:3.048}

[0,4,0] [0,0,0]     vpipe   nrml    pipe_lp_14  {’v1’:1}    {’r’:0.0254,’l’:6.096}

[4,4,0] [4,2,0]     pipe    nrml    bps_lp_1    {}          {’l’:6.096}

[4,2,0] [4,0,0]     pipe    nrml    bps_lp_2    {}          {’l’:6.096}

[0,0,2] [1,0,2]     pipe    nrml    pipe_hp_1   {}          {’l’:3.048}

[1,0,2] [2,0,2]     vpipe   nrml    pipe_hp_2   {’v1’:1}    {’r’:0.0254,’l’:6.096}

[2,0,2] [4,0,2]     vpipe   nrml    pipe_hp_3   {’v1’:1}    {’r’:0.0254,’l’:6.096}

[4,0,2] [5,0,2]     pipe    nrml    pipe_hp_4   {}          {’l’:3.048}

[5,0,2] [6,0,2]     pipe    nrml    pipe_hp_5   {}          {’l’:6.096}

[6,0,2] [7,0,2]     pipe    nrml    pipe_hp_6   {}          {’l’:3.048}

[7,0,2] [7,4,2]     vpipe   nrml    pipe_hp_7   {’v1’:1}    {’r’:0.0254,’l’:6.096}

[7,4,2] [6,4,2]     pipe    nrml    pipe_hp_8   {}          {’l’:3.048}

[6,4,2] [5,4,2]     pipe    nrml    pipe_hp_9   {}          {’l’:6.096}

[5,4,2] [4,4,2]     pipe    nrml    pipe_hp_10  {}          {’l’:3.048}
```

```
[4,4,2] [3,4,2]     pipe    nrml    pipe_hp_11  {}          {’l’:3.048}
[3,4,2] [2,4,2]     vpipe   nrml    pipe_hp_12  {’v1’:1}    {’r’:0.0254,’l’:6.096}
[2,4,2] [0,4,2]     pipe    nrml    pipe_hp_13  {}          {’l’:3.048}
[0,4,2] [0,0,2]     vpipe   nrml    pipe_hp_14  {’v1’:1}    {’r’:0.0254,’l’:6.096}
[4,4,2] [4,2,2]     pipe    nrml    bps_hp_1    {}          {’l’:6.096}
[4,2,2] [4,0,2]     pipe    nrml    bps_hp_2    {}          {’l’:6.096}
[2,4,1.8] [2,4,2]   vpipe   nrml    out_pc1     {’v1’:1}    {’r’:0.01905,’l’:0.6096}
ref@[2,4,1] [2,4,1.8] cp,0,snk,0 src pc1_out  {’v1’:1,’v2’:0,’ps1’:200,’ps2’:0}   {}
[2,4,0.2] ref@[2,4,1] cp,1,src,0 snk pc1_in   {}  {}
[2,4,0] [2,4,0.2]   vpipe   nrml    in_pc1      {’v1’:1}    {’r’:0.01905,’l’:0.6096}
[2,0,1.8] [2,0,2]   vpipe   nrml    out_pc2     {’v1’:0}    {’r’:0.01905,’l’:0.6096}
ref@[2,0,1] [2,0,1.8] cp,0,snk,0 src pc2_out  {’v1’:1,’v2’:0,’ps1’:0,’ps2’:0}   {}
[2,0,0.2] ref@[2,0,1] cp,1,src,0 snk pc2_in   {}  {}
[2,0,0] [2,0,0.2]   vpipe   nrml    in_pc2      {’v1’:0}    {’r’:0.01905,’l’:0.6096}
[1,0,2] [1,0,0]     svc_2v  nrml    svc1        {’v1’:1,’v2’:0} {}
[3,4,2] [3,4,1.8]   vpipe   nrml    in_svc2     {’v1’:1}    {’r’:0.0127,’l’:1.2192}
[3,4,1.8] [3,4,0.2] svc_1v  nrml    svc2        {’v1’:1}        {}
[3,4,0.2] [3,4,0]   vpipe   nrml    out_svc2    {’v1’:1}    {’r’:0.0127,’l’:1.2192}
[4,2,2] [4,2,0]     svc_2v  nrml    svc3        {’v1’:1,’v2’:0} {}
[5,0,2] [5,0,1.8]   vpipe   nrml    in_svc4     {’v1’:1}    {’r’:0.0127,’l’:1.2192}
[5,0,1.8] [5,0,0.2] svc_2v  nrml    svc4        {’v1’:1,’v2’:0} {}
[5,0,0.2] [5,0,0]   vpipe   nrml    out_svc4    {’v1’:1}    {’r’:0.0127,’l’:1.2192}
[5,4,2] [5,4,1.8]   vpipe   nrml    in_svc5     {’v1’:1}    {’r’:0.0127,’l’:1.2192}
[5,4,1.8] [5,4,0.2] svc_2v  nrml    svc5        {’v1’:1,’v2’:0} {}
[5,4,0.2] [5,4,0]   vpipe   nrml    out_svc5    {’v1’:1}    {’r’:0.0127,’l’:1.2192}
[6,0,2] [6,0,0]     svc_2v  nrml    svc6        {’v1’:1,’v2’:0} {}
[6,4,2] [6,4,0]     svc_2v  nrml    svc7        {’v1’:1,’v2’:0} {}
```

B.2.2 comb_anal.py

```python

import sys
sys.path.append('C:\\My_Data\\sharefolder\\mythesis-python\\src\\my_modules')
```

```
from cen_model import *
from simul import *
from control import *
import components as comps
from post_proc import *
from numpy import *
import cPickle
import copy
import time


def fac_acc(nlevel,prev_mat=None):
    '''
    This function is used in genFFD function as a subroutine for creating
    a full factorial design
    '''
    if nlevel is not 0:
        if prev_mat == None:
            new_mat = arange(nlevel)
            new_mat.shape=(nlevel,1)
        else:
            m = prev_mat.shape[0]
            new_mat = None
            for i in arange(nlevel):
                new_vec = ones(m)*i
                tran_mat = insert(prev_mat,0,new_vec,axis=1)
                if new_mat == None:
                    new_mat = tran_mat
                else:
                    new_mat = append(new_mat,tran_mat,axis=0)
    else:
        new_mat = prev_mat
    return new_mat

def genFFD(nlevel):
    '''
    Creates a full factorial experimental design
    Inputs:
        nlevel = n-dim array or list. Each element is the number
            of levels ofeach factor
    Outputs:
        Full factorial design (numpy array-type)
    '''
    if isinstance(nlevel,ndarray) == False:
        nlevel=array(nlevel)
    ffd = None
    for i in xrange(nlevel.shape[0]-1,-1,-1):
        ffd = fac_acc(nlevel[i],ffd)
    return ffd

def bin_combination(n_digit):
    '''
    This function create a list of full factorial binary combinations.
    Inputs:
        n_digit = number of digits of the binary combination.
    Output:
        List of n-digit, full factorial binary combinations.
        The list is organized as the Python dictionary format, based
        on the number of 1s in the combinations.
        ex)
            Output[2] = list of the combinations with two 1s.
    '''
    bin_list = genFFD(2*ones(n_digit))
    cmb_dic = {}
    for i in xrange(n_digit+1):
        cmb_dic[i]=[]

    for a in bin_list:
        n_valve = sum(a)
        cmb_dic[n_valve]+=[list(a)]

    return cmb_dic



################################ Script main. ################################


# Generate a library of component model functions.

c_lib = dict()
# Svc load with a single valve:
c_lib['svc_1v']=comps.svc_1v

# Svc load with double valves:
c_lib['svc_2v']=comps.svc_2v

# Chiller-pump unit:
c_lib['cp']=comps.cp

# Pipe without valve:
c_lib['pipe']=comps.pipe

# Pipe with valve:
c_lib['vpipe']=comps.vpipe

```

```
# Generate the full factorial binary combinations.
cmb_dic = bin_combination(7)
n_valve_set = [2,3,4,5,6]      # Valve amounts investigated in analysis


# bin_set represents the following pipes in order:
# ['pipe_4','pipe_5','pipe_9','pipe_10','pipe_11','bps_1','bps2']
#     0        1        2        3         4        5       6
#
# The indices of the edge elements in the focused area
# (see Figure 63) are given as follows.
e_id = [3,4,8,9,10,14,15]

# Create the result data storage.
result = {}
for n_valve in n_valve_set:
    result[n_valve]=[]


for n_valve in n_valve_set:

    for set_id, bin_set in enumerate(cmb_dic[n_valve]):
        #---------------------------------------------------------------
        # Creation of a graph model
        comp_ts = time.clock()

        # Feed the 'yp_bare_model.txt' file and the component libary
        # to create a bare graph model.
        mdl = GraphModel('yp_bare_model.txt', c_lib)

        # Reconfigure the model ------------------------------------

        e = mdl.edge['nrml']     # Call all normal edges
        for b,v in enumerate(bin_set):
            i=e_id[b]
            j=i+16

            # change a PIPE edge to a VPIPE edge based on
            # the binary string, bin_set.
            if v == 1:
                e[i].model = comps.vpipe
                e[j].model = comps.vpipe
                e[i].cv['v1']=1
                e[j].cv['v1']=1
                e[i].res_dat['v1']=[]
                e[j].res_dat['v1']=[]
                if 'bps' in e[i].name:
                    e[i].prm['r']=0.01905
                    e[j].prm['r']=0.01905
                else:
                    e[i].prm['r']=0.0254
                    e[j].prm['r']=0.0254

        # ---------------------------------------------------------------------

        # Initialize the node pressure.
        n_nodes = mdl.A.shape[0]
        ini_np = 135000*ones(n_nodes)
        mdl.set_np(ini_np)

        dt = 0.05             # time step of simulation.
        slvr = Solver(mdl,dt,1e-7,5)  # Initialize solver object.

        # Run simulation to obtain initial conditions
```

```
    for i in xrange(70):
        slvr.run_to_next_time_step()
    mdl.empty_data()                 # Reset the data storage.
    mdl.t=0                          # Reset the simulation time.
    ini_mdl = cPickle.dumps(mdl,1)   # Store the initial model.

    #---------------------------------------------------------------
    # Setup damage simulation.

    t_f = 10              # Final time of simulation.
    d_rad = 0.7           # Damage bubble radius.
    t_dmg = 1             # Time at whcih a damage occurs.

    # Create damage set.
    dmg_set.damage_set=array([[3.5,0,2],[5.25,0,2],[7,0,2],[3.5,4,2],\
    [5.25,4,2],[7,4,2],[4,1,2],[4,3,2],[7,2,2]])
    dmg_set.size = dmg_set.damage_set.shape[0]

    # Setting default component model and parameters for damaged
    # components.
    dflt_model = c_lib['pipe']
    dflt_prm = {'l':0.5,'r':0.01905}
    rupt_press = 110000      # Pressure setting for the damage nodes

    # Initialize data recorder
    recorder = Recorder()
    time_dat = zeros(dmg_set.size)

    #---------------------------------------------------------------
    # Damage simulation body:
    for i in xrange(dmg_set.size):
        print 'Case␣with␣n_valve:␣'+str(n_valve)+',␣sim␣no.:'+str(i)
```

```python
            comp_ts = time.clock()
            mdl_2 = cPickle.loads(ini_mdl)   # Load the initial graph model
            slvr.set_model(mdl_2)            # Connect the model to solver

            # Create the reference damage control model
            dcs = DmgCtrlSys()
            dcs.setup_cp_agents(mdl_2,dt)
            dcs.setup_smart_valves(mdl_2,dt)
            dcs.connect(mdl_2)

            # Update controllers in the model.
            for ag in dcs.agent:
                ag.update_self()
            for ag in dcs.agent:
                ag.update_ports()

            # Load damage bubble from damage set.
            dmg_bbl = dmg_set.go_to_set(i)
            dmg_bbl.dflt_model = dflt_model
            dmg_bbl.dflt_prm = dflt_prm

            # Connect the bubble and data recorder objects to the solver.
            slvr.dmg_bubble = dmg_bbl
            slvr.recorder = recorder

            # Run simulation until the final time t_f.
            t = 0
            while  t <= t_f:
                if slvr._conv_flag == 1:
                    for ag in dcs.agent:
                        ag.update_self()
                    for ag in dcs.agent:
                        ag.update_ports()
                    for ag in dcs.agent:
                        ag.process()
                slvr.run_to_next_time_step()
                t +=dt
            comp_t = (time.clock()-comp_ts)

            # Store the entire model into the data recorder object.
            recorder.stack_model(mdl_2)
            recorder.stack_res()
            print i,comp_t
            time_dat[i] = comp_t

        # Store results in the result data storage.
        ocr_dat=recorder.op_capa_rate(1)
        ave_ocr=mean(ocr_dat[1])

        recorder.save_res_dat('v'+str(n_valve)+'_n'+str(set_id))
        result[n_valve]+=[[bin_set,ocr_dat[1],ave_ocr,time_dat]]

# Save the entire result data in to a res. file.
result['dmg_set']=dmg_set.damage_set
f=open('result.res','w')
cPickle.dump(result,f)
f.close()
```

B.3 my_modules *Package*

B.3.1 cen_model.py

```
from numpy import *
from numpy.linalg import norm
from components import *
import copy
import sys
```

```

class Node:
    '''
    .pos     : topological position of the node
    .np      : node potential
    .id      : node id
    .dist    : topological distance from the origin point
    '''

    def __init__(self,type='nrml',pos=None,np=150000):
        self.type = type      # Node type
        self.pos = pos        # Topological coord. of node
        self.np = np          # Node potential value
        self.id = 0           # Node index
        self.set_dist()       # Distance from the origin
        self.res_dat = []     # Local data storage during simulation


    def set_dist(self):
        if self.pos != None:
            self.dist = norm(self.pos)
        else:
            self.dist = None

    def stack_res(self):
        # Store the result of the node potential during simulation.
        self.res_dat += [self.np]



class Edge:

    def __init__(self,type='nrml', node_in=Node(),node_out=Node(),\
                c_model = None,name=None):

        self.type = type           # Edge type
        self.node_in = node_in     # In-node object reference
        self.node_out = node_out   # Out-node object reference
        self.model = c_model       # Component function reference
        self.way_pt = None         # Not used currently.
        self.name = name           # Edge name (string).
        self.id = 0                # Index of an edge
        self.get_pos()             # Compute topological coord. of edge
        self.xt = 0                # State variable in t
        self.xt_1 = 0              # State variable in t-1
        self.cv = {}               # Control variables (dict type)
        self.prm = {}              # Parameters (dict type)
        self.res_dat = {'xt_1':[],'ep':[]}  # Local storage for sim result

    def __call__(self,xt_1=None,bc=None,cv=None,prm=None):
        '''
        Computes the state value of the model at the current simulation time.
        Inputs:
            xt_1  :(float) State value at T-1
            bc    :(float) Boundary condition. It is usually an edge potential
                    in an edge component.
            cv    :(float dict) Control variables.
        Output:
            (float) The current model state.
        '''
        if xt_1==None: xt_1=self.xt_1
        if bc==None: bc=self.ep
        if cv==None: cv=self.cv
        if prm==None: prm=self.prm
        xt = self.model(xt_1,bc,cv,prm)
        return xt
        ''' '''
    def get_ep(self):
        # Compute edge potential.
        ep = self.node_in.np-self.node_out.np
        return ep

    ep = property(fget = get_ep)

    def get_pos(self):
        # Compute topological position.
        if self.node_in.pos != None:
            if self.node_out.pos != None:
                self.pos = (self.node_in.pos+self.node_out.pos)/2
            else:
                self.pos = self.node_in.pos
        else:
            if self.node_out.pos != None:
                self.pos = self.node_out.pos
            else:
                self.pos = None

    def set_init_cv(self,init_cv):
        # set initial values to control variables.
        self.cv = init_cv
        for k in init_cv.keys():
            self.res_dat[k] = []

    def set_init_prm(self,init_prm):
        # Set initial values to parameters.
        self.prm = init_prm
    def set_node_in(self,node_in):
        # Set in-node.
        try:
            isinstance(node_in,Node)
        except:
            print 'node_in␣should␣be␣the␣Node␣class.'
            import sys
            sys.exit(1)

        self.node_in = node_in
        self._get_pos()

    def set_node_out(self,node_out):
        # set out-node.
        try:
            isinstance(node_out,Node)
        except:
            print 'node_out␣should␣be␣the␣Node␣class.'
            import sys
            sys.exit(1)

        self.node_out = node_out
        self._get_pos()

    def stack_res(self):
        # Store the result of edge properties during simulation.
        # Stores x(t-1) and edge potential.
        self.res_dat['xt_1'] += [self.xt_1]
        self.res_dat['ep'] += [self.ep]
        for k,v in self.cv.items():
            self.res_dat[k] += [v]

    def get_flow(self,**kwargs):
        # Compute x(t) using component model
        '''
        Inputs:
        **kwargs : the dictionary 'key=value' pairs for defining control
                    variables
        Outputs:
        The 'flow' value at time t. The flow value is also stored in the
        property self.flow.
        '''
        if kwargs != {}:
            self.cv.update(kwargs)
        self.xt = self()

class CoupledEdge(Edge):

    def __init__(self,type='nrml', node_in=Node(),node_out=Node(),\
                 c_model = None,cpl_idx=0,name=None):
        Edge.__init__(self,type,node_in,node_out,c_model,name)
        self.cpl_idx = cpl_idx      # Indexing of the inter-coupled edges
        self.couple = []            # List of the inter-coupled edges

    def __call__(self,xt_1=None,bc=None,cv=None,prm=None):
        '''
        Computes the state value of the model at the current simulation time.
        Inputs:
            xt_1 :(float) State value at T-1
            bc   :(float) Boundary condition. It is usually an edge potential
                    in an edge component.
            cv   :(float dict) Control variables.
        Output:
            (float) The current model state.
        '''
        if xt_1==None: xt_1=self.xt_1
        if bc==None: bc=self.ep
        if cv==None: cv=self.cv
        if prm==None: prm=self.prm

        xt_1_all=[]
        bc_all = []
        cv_all={}
        prm_all = {}
        xt_1_all.insert(self.cpl_idx,xt_1)
        bc_all.insert(self.cpl_idx,bc)
        cv_all.update(cv)
        prm_all.update(prm)
        for a_comp in self.couple:
            xt_1_all.insert(a_comp.cpl_idx,a_comp.xt_1)
            bc_all.insert(a_comp.cpl_idx,a_comp.ep)
            cv_all.update(a_comp.cv)
            prm_all.update(a_comp.prm)
        xt = self.model(xt_1_all,bc_all,cv_all,prm_all)[self.cpl_idx]
        return xt

class GraphModel:
    ''' '''
    def __init__(self,file=None,c_lib=None):
        self.A_tot = 0      # Complete incidence matrix
        self.A = 0          # Reduced incidence matrix
        self.L = 0          # Laplace matrix
        self.q = 0          # Flow vector
        self.S = 0          # Source flow vector
        self.R = 0          # Rupture flow vector
        # Storage of edge object references.
        self.edge ={'nrml':[], 'dmg':[], 'src':[], 'snk':[]}
        # Storage of node object references.
        self.node ={'nrml':[], 'dmg':[], 'ref':[]}
        self.t = 0          # Sim time
        self.t_dat = []     # Sim time data

        if file is not None:    # Check if component library is given.
            try:
                c_lib is None
            except:
                print "You should provide c_lib as an input."
            self.c_lib = c_lib
            self.create_from_data(file)


    def create_from_data(self, file):
        '''
        Create graph components based on the initialization file.
        .file : Data file name (string)
        '''
        couple_stack={}
        f = open(file, 'r')
        words = f.readline().split()
        data_type = 0
        while words != []:
            # Iteration routine:

            # first choose what to do.
            if '#' in words[0]:
                data_type = 0
                pass
            elif words[0] == 'edge:':
                data_type = 1
                words=f.readline().split()
            elif words[0] == 'x_bound:':
                data_type = 0
                self.x_bound = [float(words[1]),float(words[2])]
            elif words[0] == 'y_bound:':
                data_type = 0
                self.y_bound = [float(words[1]),float(words[2])]
            elif words[0] == 'z_bound:':
                data_type = 0
                self.z_bound = [float(words[1]),float(words[2])]

            # Then do the assignment.

            if data_type == 1:
                node = []
                for i in xrange(2):
                    if '@' in words[i]:
                        node_info = words[i].split('@')
                        type = node_info[0]
                        pos = array(eval(node_info[1]),dtype=float)
                        if type == 'ref':
                            a_node = Node(type,pos,0)
                        elif type == 'dmg':
                            a_node = Node(type,pos,110000)
                        elif type == 'nrml':
                            a_node = Node(type,pos)
                    else:
                        type = 'nrml'
                        pos = array(eval(words[i]),dtype=float)
                        a_node = Node(type,pos)
                    node.append(a_node)
                if ',' in words[2]:
                    edge_info = words[2].split(',')
                    an_edge = CoupledEdge(words[3], node[0], node[1],\
                            self.c_lib[edge_info[0]],eval(edge_info[1])\
                            , words[4])
                    couple_stack[an_edge] = "self.edge['"+edge_info[2]+"']\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣["+edge_info[3]+"]"
                else:
                    an_edge = Edge(words[3],node[0],node[1],\
                            self.c_lib[words[2]],words[4])
                an_edge.set_init_cv(eval(words[5]))
                an_edge.set_init_prm(eval(words[6]))
                self.add_edge(an_edge)

            # End of iteration routine
            words = f.readline().split()
        else:
            f.close()
            for key,value in couple_stack.items():
                key.couple.append(eval(value))
            self.update_model()


    def info(self,which):
        '''
        Gives info. of graph components.
        Input:
            single string among 'edge','node','source','sink', and'damage'
        Output:
            A list of the corresponding objects with a few useful info.
        '''
        if which == 'node':
            for type in ['nrml','ref','dmg']:
                print type
                for i in xrange(len(self.node[type])):
                    print i, self.node[type][i].pos, self.node[type][i].np
        elif which == 'edge':
            for type in ['nrml','dmg','src','snk']:
                print type
                for i in xrange(len(self.edge[type])):
                    an_edge = self.edge[type][i]
                    print i, an_edge.pos, an_edge.model, an_edge.name\
                    ,an_edge.xt_1 , an_edge.ep

    def get_edge_dict(self):
        # Gives the edge object references in the dict data type.
        # keyword is the name of an edge.
        edges = self.edge
        edge_list = edges['nrml']+edges['src']+edges['snk']+edges['dmg']
        edge_dict = {}
        for an_edge in edge_list:
            edge_dict[an_edge.name]=an_edge
        return edge_dict

    def get_node_dict(self):
        # Gives the node object references in the dict data type.
        # keyword is the topological coord. of a node.
        nodes = self.node
        node_list = nodes['nrml']+nodes['ref']+nodes['dmg']
        node_dict ={}
        for a_node in edge_list:
            node_dict[a_node.pos]=a_node
        return node_dict

    def setA(self):
        # Update the A_tot
        edges = self.edge
        nodes = self.node
        edge_list = edges['nrml']+edges['src']+edges['snk']+edges['dmg']
        node_list = nodes['nrml']+nodes['ref']+nodes['dmg']
        self.A_tot=zeros((len(node_list),len(edge_list)),dtype=integer)
        for i in xrange(len(edge_list)):
            m = node_list.index(edge_list[i].node_in)
            n = node_list.index(edge_list[i].node_out)
            self.A_tot[m,i]=1
            self.A_tot[n,i]=-1
        self.A = self.A_tot[:len(nodes['nrml']),:]



    def setq(self):
        # Update the flow vector.

        edges = self.edge
        edge_list = edges['nrml']+edges['src']+edges['snk']+edges['dmg']
        self.q = array([an_edge.xt for an_edge in edge_list])
        # self.q_tot = array([an_edge.xt for an_edge in edge_list])
        # self.q = q_tot[:len(edges['nrml'])]


    def setS(self):
        # Update the source/sink vector
        n_nodes = len(self.node['nrml'])
        self.S = zeros(n_nodes)
        for a_src in self.edge['src']:
            self.S[a_src.node_out.id] = a_src.xt
        for a_snk in self.edge['snk']:
            self.S[a_snk.node_in.id] -= a_snk.xt


    def setR(self):
        # Update the rupture flow vector
        dmg_list = self.edge['dmg']
        self.R = zeros(len(self.node['nrml']))
        for a_dmg in dmg_list:
            if a_dmg.node_out.type == 'dmg':
                self.R[a_dmg.node_in.id]=-a_dmg.xt
            else:
                self.R[a_dmg.node_out.id]=a_dmg.xt

    def setL(self):
        # Update Laplace matrix
        self.L = inner(self.A, self.A)


    def update_model(self):
        # Update all.
        self.setA()
        self.setL()
        self.setq()
        self.setS()
        self.setR()

    def get_np(self):
        # Extract the node potentials in the graph model
        np = array([nd.np for nd in self.node['nrml']])
        return np

    def set_np(self,vec):
        # Set node potential values in the graph model
        node_list = self.node['nrml']
        for i in xrange(len(node_list)):
            node_list[i].np = vec[i]


    def get_ep(self):
        # Obtain all edge potentials in the graph.
        ep = array([eg.ep for eg in self.edge['nrml']])
        return ep

    def empty_data(self):
        # Reset data storage.
        nodes = self.node
        edges = self.edge
        for a_node in nodes['nrml']+nodes['ref']+nodes['dmg']:
            a_node.res_dat = []
        for an_edge in edges['nrml']+edges['src']+edges['snk']+edges['dmg']:
            for k in an_edge.res_dat.iterkeys():
                an_edge.res_dat[k] = []
        self.t_dat = []


    def add_edge(self,an_edge):
        # Add an edge to graph model
        node_list = []
        for k in self.node.keys():
            node_list += self.node[k]

        for a_node in node_list:
            if (an_edge.node_in.pos == a_node.pos).all():
                an_edge.node_in = a_node
                break
        else:
            self.add_node(an_edge.node_in)


        for a_node in node_list:
            if (an_edge.node_out.pos == a_node.pos).all():
                an_edge.node_out = a_node
                break
        else:
            self.add_node(an_edge.node_out)

        self.edge[an_edge.type] += [an_edge]
        self.edge[an_edge.type][-1].id = len(self.edge[an_edge.type])-1
        self.update_model()



    def remove_edge(self,an_edge):
        # Remove an edge in graph model
        '''
        The edge to be removed is assumed to be one of the edge objects
        in a graphModel object (e.g. A.edge).
        '''
        edge_list=self.edge[an_edge.type]
        edge_list.remove(an_edge)
        for i in xrange(len(edge_list)):
            edge_list[i].id = i
        self.update_model()

        node_list=self.node[an_edge.node_in.type]
        orph_node_chk = node_list.index(an_edge.node_in)
        if (self.A[orph_node_chk,:]==0).all():
            node_list.remove(an_edge.node_in)
            for i in xrange(len(node_list)):
                node_list[i].id = i

        node_list=self.node[an_edge.node_out.type]
        orph_node_chk = node_list.index(an_edge.node_out)
        if (self.A[orph_node_chk,:]==0).all():
            node_list.remove(an_edge.node_out)
            for i in xrange(len(node_list)):
                node_list[i].id = i
        self.update_model()
```

B.3.2 components.py

```
from numpy import *
import copy

def tramnmx(x,x_min,x_max):
    return 2*(x - x_min)/(x_max - x_min)-1

def postmnmx(yn,y_min,y_max):
    return (yn+1)/2*(y_max-y_min)+y_min



def svc_2v(xt_1,bc,cv,prm):
    x = array(xt_1,dtype=float)
    x = append(x,bc)
    x = append(x,[cv['v1'],cv['v2']])
    try:
        svc_2v.b1
    except:
```

```
svc_2v.b1=array([-0.403177952467408,])
svc_2v.iw1=array([[0.643235935305463,0.33278775193152,],\
    ])
svc_2v.b2=array([-0.411336700943652,-0.22611450035137,\
-8.42624415802863,-4.57147842510292,-0.330952340870594,\
0.286029916201246,9.51305573449424,-1.3624903798612,\
-0.995320877580977,0.537985803263872,])
svc_2v.iw2=array([[0.0714896740141171,0.124308795738995,],\
    [0.0242441954149286,-0.242538338148743,],\
    [-4.55392432505877,-4.66960053358728,],\
    [-1.43335059359974,-1.53166556026716,],\
    [0.0708805796681024,0.125638424118948,],\
    [0.183070644737504,-0.056157845478681,],\
    [5.2761727358014,5.39474034211337,],\
    [0.084608131045956,0.147372301198779,],\
    [1.0506297378929,-11.9293135157161,],\
    [-0.0760006138515422,-0.126590360533179,],\
    ])
svc_2v.lw1=array([[-3.4462330348185,],\
    [-0.542191481408433,],\
    [0.80755937725042,],\
    [-1.01341095849417,],\
    [-2.83974604045263,],\
    [0.50754871664009,],\
    [-0.801381796064937,],\
    [-1.20213023705471,],\
    [-5.74208780513262,],\
    [4.4153876027629,],\
    ])
svc_2v.b3=array([-0.453891928155915,])
svc_2v.lw2=array([[2.94936641864294,-0.501689548727145,\
-4.33470770532195,0.433940266592169,-2.63198105874181,\
        0.372434579799262,-3.57198182181452,-0.636037498946495,\
        0.00108017389824621,0.758199584717331,],\
            ])
        svc_2v.x_min=array([-0.000299796255631329,-50000,0,0,])
        svc_2v.x_max=array([0.000299943261161827,100000,1,1,])
        svc_2v.y_min=array([-0.000299933385146379,])
        svc_2v.y_max=array([0.000299985845887199,])

    xn = tramnmx(x,svc_2v.x_min,svc_2v.x_max)

    y_l1 = inner(svc_2v.iw1,xn[:2])+svc_2v.b1
    y_l2 = tanh( inner(svc_2v.lw1,y_l1)+inner(svc_2v.iw2,xn[2:])+svc_2v.b2 )
    y = inner(svc_2v.lw2,y_l2)+svc_2v.b3
    return postmnmx(y,svc_2v.y_min,svc_2v.y_max)[0]

def svc_1v(xt_1,bc,cv,prm):
    x = array(xt_1,dtype=float)
    x = append(x,bc)
    x = append(x,[cv['v1']])
    try:
        svc_1v.b1
    except:
        svc_1v.b1=array([-0.583122932082598,])
        svc_1v.iw1=array([[0.37656113493791,0.165164324632059,],\
            ])
        svc_1v.b2=array([-1.45705456462713,4.63915544098338,\
        -3.44660278816248,0.36680376394936,6.82793180652776,\
        0.368368065322034,])
        svc_1v.iw2=array([[-1.59988528730233,],\
            [0.334543746249295,],\
            [0.049274148148873,],\
            [-0.412887536557245,],\
            [9.70446015255161,],\
            [-0.660781163120365,],\
            ])
        svc_1v.lw1=array([[-1.08660407076284,],\
            [-3.79312535007853,],\
            [-5.94497899956459,],\
            [0.0487255974882004,],\
            [-6.54974804274007,],\
            [-0.291478251193739,],\
            ])
        svc_1v.b3=array([0.270258273388033,])
        svc_1v.lw2=array([[-0.546132517009071,-1.1395492938795,\
        -0.11454017366658,8.19432738793683,0.327227515281873,\
        -5.0716205960296,],\
            ])
        svc_1v.x_min=array([-0.000499407951209851,-100000,0,])
        svc_1v.x_max=array([0.000497025935259093,100000,1,])
        svc_1v.y_min=array([-0.000499346034219654,])
        svc_1v.y_max=array([0.000499771370327245,]}

    xn = tramnmx(x,svc_1v.x_min,svc_1v.x_max)

    y_l1 = inner(svc_1v.iw1,xn[:2])+svc_1v.b1
    y_l2 = tanh( inner(svc_1v.lw1,y_l1)+inner(svc_1v.iw2,xn[2:])+svc_1v.b2 )
    y = inner(svc_1v.lw2,y_l2)+svc_1v.b3
    return postmnmx(y,svc_1v.y_min,svc_1v.y_max)[0]


def cp(xt_1,bc,cv,prm):
    x = array(xt_1,dtype=float)
    x = append(x,bc)
    x[2]=-x[2]
    x = append(x,[cv['v1'],cv['ps1']])
    try:
        cp.b1
    except:
        cp.b1=array([0.0489904381537137,1.13644075211915,])
        cp.iw1=array([[-0.525796183426231,0.0192782911616325,\
        0.154217122353081,-0.00306133003894085,],\
            [0.615191748886452,0.102512901557262,-0.182102525499998,\
            0.0214476072620936,],\
            ])
        cp.b2=\
        array([-1.89204471174595,0.0344059476555241,\
        -0.0320653909948023,1.51288680922981,-3.6519977277266,\
        -3.63131551527321,3.65276359754262,0.22248668152001,\
        -3.65074623502132,1.50454827855458,])
        cp.iw2=array([[0.0178131489728337,-1.12123711514336,],\
            [0.234357275896735,0.785708489179496,],\
            [0.0522225021503697,0.852473149704312,],\
            [-0.0328117710483785,-0.651051476301202,],\
            [-3.92429546617008,0.498279864303038,],\
            [-4.09534057873157,0.638523158404742,],\
            [0.000279275082042311,0.0123095457831633,],\
            [-0.11481872672073,0.734734894682054,],\
            [-3.02051106223508,0.306505472915975,],\
            [-0.0647526268147423,1.66988409081991,],\
            ])
        cp.lw1=array([[3.43329806921937,-2.24616094270238,],\
            [-1.09768448721661,0.736039661516823,],\
            [-1.18830768627075,0.872904017923657,],\
            [0.360700852490147,-0.388997731228925,],\
            [-0.612338007383116,-0.0403040667495736,],\
            [-0.979852328329821,-0.0815320390826407,],\
            [-3.79013945739699,-3.19467304486502,],\
            [-1.04713271264814,0.791836836443043,],\
            [-0.285115896429936,0.0866149372692415,],\
            [-1.33308891945178,0.469298202473634,],\
            ])
        cp.b3=array([-2.80838100219508,-0.442879938961615,])
        cp.lw2=array([[0.335115175380668,1.63575209456726,-2.68140339429855,\
        -0.951733966917486,3.31551908986134,-1.82312457170654,\
        0.0426994969396584,2.49409270219835,-4.53980654966542,\
        -0.470449369827458,],\
            [-0.0125660189890549,0.0198181499835772,0.0100659482519615,\
            -0.13129802166065,0.235816958593532,-0.151630742298007,\
            -2.21808254323842,0.0445207304417349,-0.253936436139287,\
            -0.0368347385572102,],\
            ])
        cp.x_min=array([-1.32963267830196e-010,-0.000908858231239424,\
        100000,100000,0,0,])
        cp.x_max=array([0.00124845914803711,0.00189215940887209,450000,\
        250000,1,400,])
        cp.y_min=array([-1.32963267830196e-010,-0.000908858231239424,])
        cp.y_max=array([0.00124845914803711,0.00189215940887209,])
    xn = tramnmx(x,cp.x_min,cp.x_max)


    y_l1 = inner(cp.iw1,xn[:4])+cp.b1
    y_l2 = tanh( inner(cp.lw1,y_l1)+inner(cp.iw2,xn[4:])+cp.b2 )
    y = inner(cp.lw2,y_l2)+cp.b3
    if x[4] <=0.1:
        ys = postmnmx(y,cp.y_min,cp.y_max)*exp(69.08/4*(x[4]-0.1))
    else:
        ys = postmnmx(y,cp.y_min,cp.y_max)
    return ys
    return postmnmx(y,cp.y_min,cp.y_max)


def pipe(xt_1,bc,cv,prm):
    x = array(xt_1,dtype=float)
    x = append(x,bc)
    x = append(x,[prm['l']])
    if x[-1] < 0.7:
        x[-1] = 0.7
    try:
        pipe.b1
    except:
        pipe.b1=array([-0.722645087070832,])
        pipe.iw1=array([[-0.43779009279619,-0.044495749083984,],\
            ])
        pipe.b2=array([-0.500371857384726,0.800020085270251,\
        0.493695739179278,-4.45917340364765,3.22203851680239,])
        pipe.iw2=array([[-0.127324154499375,],\
            [-0.675769286561483,],\
            [-0.429313396883844,],\
            [1.009556491894,],\
            [0.764809439527272,],\
            ])
        pipe.lw1=array([[-0.8544646246795,],\
            [0.978338088341976,],\
            [0.674978688777447,],\
            [-5.07643947281922,],\
            [2.44809362002147,],\
            ])
        pipe.b3=array([-0.154527122875052,])
        pipe.lw2=array([[2.01546446805134,0.421382595252777,\
        -1.26605905668604,0.0140161229334608,-0.114816716873872,],\
            ])
        pipe.x_min=array([-0.000999731838454432,-10000,0.5,])
        pipe.x_max=array([0.000998936066170416,10000,5,])
        pipe.y_min=array([-0.000999922140080304,])
        pipe.y_max=array([0.000999070607852158,])

    xn = tramnmx(x,pipe.x_min,pipe.x_max)

    y_l1 = inner(pipe.iw1,xn[:2])+pipe.b1
    y_l2 = tanh( inner(pipe.lw1,y_l1)+inner(pipe.iw2,xn[2:])+pipe.b2 )
    y = inner(pipe.lw2,y_l2)+pipe.b3
    return postmnmx(y,pipe.y_min,pipe.y_max)[0]

def vpipe_127(x):
    if x[-1] < 0.5:
        x[-1] = 0.5
    try:
        vpipe_127.b1
    except:
        vpipe_127.b1=array([-0.0510665578773549,])
        vpipe_127.iw1=array([[0.959585939020714,0.700380369079835,],\
            ])
        vpipe_127.b2=array([2.22111219764168,1.15032392308498,\
        -0.0964031438028126,0.310465730214223,-1.02912528084642,\
        1.06778583889189,-0.931426360825884,])
        vpipe_127.iw2=array([[-1.00767842344865,-0.112610634234768,],\
            [-0.478339787605095,-0.00134991663890141,],\
            [0.631117908111167,0.0875647409550213,],\
            [0.333505543758612,-0.0974041743992097,],\
            [0.649476953620522,-0.0126161444606362,],\
            [-0.797920885590477,0.0354739773511861,],\
            [-0.492689606360934,0.0589309609776407,],\
            ])
        vpipe_127.lw1=array([[0.604002566769382,],\
            [0.0368488663710622,],\
            [0.618445221091317,],\
            [5.76390087348398,],\
            [0.140766399515174,],\
            [1.29165239352796,],\
            [1.05042804425546,],\
            ])
        vpipe_127.b3=array([-5.95790273363427,])
        vpipe_127.lw2=array([[-1.47977898763016,14.9880477399079,\
        0.675066167589956,0.0448182808629973,6.47685142504861,\
        -0.229431256861141,-0.580339970601853,],\
            ])
        vpipe_127.x_min=array([-0.000999923830254839,-100000,0,0.3,])
        vpipe_127.x_max=array([0.000998783917717905,100000,1,1.5,])
        vpipe_127.y_min=array([-0.000999799270325252,])
        vpipe_127.y_max=array([0.000999445844846383,])

    xn = tramnmx(x,vpipe_127.x_min,vpipe_127.x_max)

    y_l1 = inner(vpipe_127.iw1,xn[:2])+ vpipe_127.b1
    y_l2 = tanh(  inner(vpipe_127.lw1,y_l1)\
                        +inner(vpipe_127.iw2,xn[2:])+ vpipe_127.b2  )
    y = inner(vpipe_127.lw2,y_l2)+vpipe_127.b3
    return postmnmx(y,vpipe_127.y_min,vpipe_127.y_max)[0]

def vpipe_1905(x):
    if x[-1] < 0.5:
        x[-1]=0.5
    try:
        vpipe_1905.b1
    except:
        vpipe_1905.b1=array([1.36875755835991,])
        vpipe_1905.iw1=array([[0.899565135459478,0.343075141096446,],\
            ])
        vpipe_1905.b2=array([4.13919295545714,2.57782245519917,\
        -2.23013427607142,1.86114219649325,0.0514370612408742,\
        11.6440644036579,-0.560040285619765,])
        vpipe_1905.iw2=array([[1.23273436664902,0.00182297874901823,],\
            [2.48370752208954,-0.910373112937128,],\
            [0.130911817895199,0.344292472771948,],\
            [0.916990899608765,0.0133814576823347,],\
            [0.0390150826101735,0.0592627481285842,],\
            [8.99263158747699,-0.00854628308211276,],\
            [-0.118751825674156,-0.206915719335347,],\
            ])
        vpipe_1905.lw1=array([[-0.968938953226134,],\
            [5.05331095660151,],\
            [2.88221279933273,],\
            [0.875250395918153,],\
            [0.285230841972612,],\
            [-2.83794761379428,],\
            [0.37115561345349,],\
            ])
        vpipe_1905.b3=array([2.38580913986778,])
        vpipe_1905.lw2=array([[3.5709882774559,3.29690419657745,\
        -0.0732992810491271,-10.6448720743607,3.6316286804983,\
        -0.0249871776473446,0.841168229489061,],\
            ])
        vpipe_1905.x_min=array([-0.000999745926433204,-100000,0,0.3,])
        vpipe_1905.x_max=array([0.000999651153384713,100000,1,6.5,])
        vpipe_1905.y_min=array([-0.0009999764105342,])
        vpipe_1905.y_max=array([0.00099989972631468,])

    xn = tramnmx(x,vpipe_1905.x_min,vpipe_1905.x_max)


    y_l1 = inner(vpipe_1905.iw1,xn[:2])+vpipe_1905.b1
    y_l2 = tanh( inner(vpipe_1905.lw1,y_l1)+\
    inner(vpipe_1905.iw2,xn[2:])+vpipe_1905.b2 )
    y = inner(vpipe_1905.lw2,y_l2)+vpipe_1905.b3
    if x[2] <=0.05:
        ys = postmnmx(y,vpipe_1905.y_min,vpipe_1905.y_max)[0]\
        *exp(69.08/2*(x[2]-0.05))
    else:
        ys = postmnmx(y,vpipe_1905.y_min,vpipe_1905.y_max)[0]
    return ys

def vpipe_254_sdp(x):
    try:
        vpipe_254_sdp.b1
    except:
        vpipe_254_sdp.b1=array([-0.0636768303109408,])
        vpipe_254_sdp.iw1=array([[-0.555741792107273,-0.0122647952398165,],\
            ])
        vpipe_254_sdp.b2=array([-0.253644812403496,-0.804912732631754,\
        0.634384497660935,0.706735508112529,1.10718223152414,\
        1.47987665977575,3.56776031360815,])
        vpipe_254_sdp.iw2=array([[0.278561878168683,0.000553426546447999,],\
            [1.78200757064552,0.412419842164154,],\
            [-0.247278136978737,-0.025678951610713,],\
            [1.58409192109964,0.370916402319805,],\
            [-0.535353546254542,-0.0901005256501447,],\
            [-0.105581278510097,0.376908095675417,],\
            [1.87625852060181,0.386659132360343,],\
            ])
        vpipe_254_sdp.lw1=array([[0.667263941699639,],\
            [-2.72753841587263,],\
            [0.666605117356617,],\
            [-1.25390363923288,],\
            [1.28085295078606,],\
            [-12.3598211701449,],\
            [-5.43824197477544,],\
            ])
        vpipe_254_sdp.b3=array([1.59381224918562,])
        vpipe_254_sdp.lw2=array([[-1.17924159357458,-0.0365516451414706,\
        -3.68629777049936,-0.0756732972648056,1.07781123264949,\
        -0.00612742506573572,-0.770369258079715,],\
            ])
        vpipe_254_sdp.x_min=array([-0.00099920532013025,-4000,0,2,])
        vpipe_254_sdp.x_max=array([0.000997215835240223,4000,1,7,])
        vpipe_254_sdp.y_min=array([-0.000973318511625282,])
        vpipe_254_sdp.y_max=array([0.000968587716533465,])
    if x[3] < 2.5:
        x[3] = 2.5
    xn = tramnmx(x,vpipe_254_sdp.x_min,vpipe_254_sdp.x_max)


    y_l1 = inner(vpipe_254_sdp.iw1,xn[:2])+vpipe_254_sdp.b1
    y_l2 = tanh( inner(vpipe_254_sdp.lw1,y_l1)+\
    inner(vpipe_254_sdp.iw2,xn[2:])+vpipe_254_sdp.b2 )
    y = inner(vpipe_254_sdp.lw2,y_l2)+vpipe_254_sdp.b3
    if x[2] <=0.05:
        ys = postmnmx(y,vpipe_254_sdp.y_min,vpipe_254_sdp.y_max)[0]\
        *exp(69.08/2*(x[2]-0.05))
    else:
        ys = postmnmx(y,vpipe_254_sdp.y_min,vpipe_254_sdp.y_max)[0]
    return ys


def vpipe_254(x):
    try:
        vpipe_254.b1
    except:
        vpipe_254.b1=array([-0.294483183501346,])
        vpipe_254.iw1=array([[-0.600385382331592,-0.424793393380238,],\
            ])
        vpipe_254.b2=array([-7.74370830861366,0.593281730027768,\
        -2.56867638630015,-0.0572371515300161,0.0328733845949659,\
        -1.82687136736938,-1.50406862149836,0.731074186500534,\
        -4.09209874680234,4.08813248576604,5.50175733945317,\
        -0.698529944446116,8.62032280386742,-8.20852861920296,\
        -8.49896203577908,])
        vpipe_254.iw2=array([[-5.41518849165587,0.00142526234507933,],\
            [-1.29919857903066,-1.8501684029622,],\
            [0.00249922603094099,-0.127311839505703,],\
            [0.0778384648131845,0.0913091215633245,],\
            [-0.0845441257440447,-0.121205694719303,],\
            [-1.64630873085545,0.0666672856002693,],\
            [1.69136606672478,-1.28758863494888,],\
            [0.699671223666673,-2.05937493355888,],\
            [-6.19641694759062,0.0193613076720055,],\
            [6.2655511279203,-0.0198760616593509,],\
            [2.73406040049057,-0.136115676218369,],\
            [-3.76445780137563,3.10934514833497,],\
            [8.82184541444878,0.0170167996372364,],\
            [-8.19162288833985,-0.0162734628490648,],\
            [-8.34746751585979,-0.0177589665973259,],\
            ])
        vpipe_254.lw1=array([[-3.47200060985354,],\
            [-2.55510247864857,],\
            [-1.12013804328657,],\
            [-0.411371739201308,],\
            [-0.660536499712842,],\
            [2.60287698445256,],\
            [1.2859730297526,],\
            [-1.62121531863431,],\
            [-0.261379873820507,],\
            [0.159535743447367,],\
            [5.07094657353168,],\
            [-3.39356494875782,],\
            [-1.65387728299081,],\
            [1.85665556993033,],\
            [2.16975511919364,],\
            ])
        vpipe_254.b3=array([1.73071246770589,])
        vpipe_254.lw2=array([[-0.265098896426814,-0.00586749879714345,\
        1.74497737674192,1.66851707634676,1.3886118566969,0.232416100685957,\
        -0.013173159420462,-0.00682479141158398,-1.03308420809935,\
        -1.00517150941908,0.0977627194777375,0.00341936350183569,\
        3.66615495730773,8.44207332103058,-4.16988224061704,],\
            ])
        vpipe_254.x_min=array([-0.000499993191012689,-70000,0,2,])
        vpipe_254.x_max=array([0.00049998369025326,70000,1,7,])
        vpipe_254.y_min=array([-0.000499989572489278,])
        vpipe_254.y_max=array([0.00049998369025326,])

    if x[3] < 2.5:
        x[3] = 2.5
    xn = tramnmx(x,vpipe_254.x_min,vpipe_254.x_max)

    y_l1 = inner(vpipe_254.iw1,xn[:2])+vpipe_254.b1
    y_l2 = tanh( inner(vpipe_254.lw1,y_l1)+\
    inner(vpipe_254.iw2,xn[2:])+vpipe_254.b2 )
    y = inner(vpipe_254.lw2,y_l2)+vpipe_254.b3

    if x[2] <=0.05:
        ys = postmnmx(y,vpipe_254.y_min,vpipe_254.y_max)[0]\
        *exp(69.08/2*(x[2]-0.05))
    else:
        ys = postmnmx(y,vpipe_254.y_min,vpipe_254.y_max)[0]
    return ys



def vpipe(xt_1,bc,cv,prm):
    x = array(xt_1,dtype=float)
    x = append(x,bc)
    x = append(x,[cv['v1'],prm['l']])
    r = prm['r']
    try:
        vpipe.d_127
    except:
        vpipe.d_127 = vpipe_127
        vpipe.d_1905 = vpipe_1905
        vpipe.d_254_sdp = vpipe_254_sdp
        vpipe.d_254 = vpipe_254

    if r > 0.0222:
        if bc < 4000 and bc > -4000:
            return vpipe.d_254_sdp(x)
        else:
            return vpipe.d_254(x)
    elif r > 0.0159:
        return vpipe.d_1905(x)
    else:
        return vpipe.d_127(x)
```

B.3.3 simul.py

```
import sys
from numpy import *
from numpy.lib.function_base import linspace, append
from numpy.linalg import norm, solve
from cen_model import Node
import copy


class DamageSet:
    '''
    Makes a list of damage scenarios for a given 3-d grid.
    .x_set, y_set, z_set = [lowerBound, UpperBound, GridDensity]
    where,
        lowerBound     : Lower bound of each coordinate of the
                         topological grid
        upperBound     : Upper bound ...
        GridDensity    : Number of grid between the lower and upper bound

    .d_Rad = radius of a damage bubble which represents a spherical damage
            region in the topological graph model
    '''

    def __init__(self,x_set,y_set,z_set,d_rad,t_dmg):
        self.x_set=linspace(x_set[0], x_set[1], x_set[2])
        self.y_set=linspace(y_set[0], y_set[1], y_set[2])
        self.z_set=linspace(z_set[0], z_set[1], z_set[2])
        self.d_rad=d_rad        # Radius of damage bubble
        self.t_dmg = t_dmg        # Damage time
        self.size=None            # Size of the damage list
        self._index = None
        self.build_sets()

    def build_sets(self):
        # Create the damage list.
        self.damage_set=zeros((1,3))
        for zPt in self.z_set:
            for yPt in self.y_set:
                for xPt in self.x_set:
                    self.damage_set = append(self.damage_set,\
                    [[xPt,yPt,zPt]],axis=0)
        self.damage_set = delete(self.damage_set,0,0)
        self.size = self.damage_set.shape[0]


    def get_next_set(self):
        # Load the next damage from the list
        if self._index is None:
            self._index=0
        else:
            self._index+=1

        a_bubble = DamageBubble(self.damage_set[self._index,:]\
        ,self.d_rad,self.t_dmg)

        return a_bubble


    def go_to_set(self,index):
        # Load the damage on a certain index of the list
        try:
            (index >= 0) and (index < self.size)
        except:
            print 'Index out of the list of damage sets.'

        self._index=index
        a_bubble = DamageBubble(self.damage_set[self._index,:],\
        self.d_rad,self.t_dmg)
        return a_bubble


    def reset_index(self):
        # Reset the current indexer to 0.
        self._index = None


class DamageBubble:
    '''
    .c_pt     : 3-dim coord. of the center of a damage bubble
    .r        : radius of the bubble
     '''
    def __init__(self,center,rad,t_dmg,dflt_model=None,dflt_prm={}):
        self.c_pt = center            # Topo. coord. of damage center
        self.r = rad                  # Damage radius
        self.t_dmg = t_dmg            # Damage time
        self.dmg_applied = False      # Flag. If damage was applied, True
        self.dflt_model = dflt_model  # Function reference of default model
        self.dflt_prm = dflt_prm      # Param of default model
        self.np_dmg = 110000          # Pressure of damage nodes


    def __call__(self,g_model):
        # g_model: graph model object reference

        '''
        pt_in       : in-node coordinate
        pt_out      : out-node coordinate
        When providing the above inputs, the function returns a boolean
        value "True" if the edge (or the part of it) is inside the bubble
        or "False" if outside it.

        The algorithm is based on a simple calculus: if a line meets a
        sphere of a damage, this means the subsitution of the line
        equation X=vec(a)+t*vec(b) into the spherical equation
        (vec(X)-vec(Xc))^T * (vec(X)-vec(Xc)) = R^2 has at least one
        solution of t with the range of 0 <= t <= 1, therefore the
        decision can be made by solving the 2nd order polynomial equation.

        If the input has only one parameter, which is pt_in, it considers
        it as an attempt to check the damage of a node. If the location of
        a node is inside the damage bubble, it gives out the boolean
        output "True."
        '''

        # Known values:
        r = self.r
        c_pt = self.c_pt
        np_dmg = self.np_dmg
        edge = g_model.edge
        node = g_model.node

        # for an_edge in edge:
        for an_edge in edge['nrml']+edge['src']+edge['snk']:
            node_in = an_edge.node_in
            node_out = an_edge.node_out
            pt_in = node_in.pos
            pt_out = node_out.pos

            # "Is it damaged" check:
            d1 = pt_in-self.c_pt
            d1_sqr = inner(d1,d1)
            d2 = pt_out-self.c_pt
            d2_sqr = inner(d2,d2)
            r_sqr = r**2

            # Line eqn(with t, the 1-D location of dir vector length):
            dir_vec = pt_out - pt_in

            # eqn of the dist btw the line and center of damage bubble
            # (a*t^2+b*t+c-r^2 = 0):
            a = inner(dir_vec,dir_vec)
            b = 2*inner(dir_vec,d1)
            c = d1_sqr - r_sqr
            D = b**2 - 4*a*c

            if D >= 0:
                t_low = (-b-D**0.5)/2/a
                t_high = (-b+D**0.5)/2/a
                if an_edge.type == 'nrml':


                    if t_low > 1 or t_high < 0:
                        # Do nothing unless an edge is inside the bubble.
                        pass

                    elif t_low <= 0 and t_high <= 1:
```

```
                # ***node_in is damaged

                # create a DamageNode
                new_pos = pt_in + t_high*dir_vec
                new_d_node = Node('dmg',new_pos,np_dmg)

                # change the length
                if an_edge.prm.has_key('l'):
                    an_edge.prm['l'] = (1-t_high)*an_edge.prm['l']
                else:
                    an_edge.prm['l'] = self.dflt_prm['l']
                if an_edge.prm.has_key('r') == False:
                    an_edge.prm['r'] = self.dflt_prm['r']

                # if there is more 50% of damage, the edge is
                # assumed to be just a pipe (default component
                # model), which doesn't have any control device.
                if t_high >= 0.5:
                    an_edge.model=self.dflt_model
                    an_edge.cv ={}
                if node_in in node[node_in.type]:
                    node[node_in.type].remove(node_in)
                an_edge.node_in = new_d_node
                node['dmg'].append(new_d_node)
                edge[an_edge.type].remove(an_edge)
                an_edge.type = 'dmg'
                an_edge.get_pos()
                edge['dmg'].append(an_edge)

            elif t_low <= 0 and t_high >= 1:
                # Entire edge damaged and disappeared

```

```
                        an_edge.model = self.dflt_model
                        an_edge.cv = {}
                        if an_edge.prm.has_key('l') == False:
                            an_edge.prm['l'] = self.dflt_prm['l']
                        if an_edge.prm.has_key('r') == False:
                            an_edge.prm['r'] = self.dflt_prm['r']
                        if node_in in node[node_in.type]:
                            node[node_in.type].remove(node_in)
                        if node_out in node[node_out.type]:
                            node[node_out.type].remove(node_out)
                        edge[an_edge.type].remove(an_edge)

                    elif t_low > 0 and t_high < 1:
                        # ****Only edge body damaged

                        another = copy.deepcopy(an_edge)
                        another.node_in = an_edge.node_in
                        another.node_out = an_edge.node_out
                        if t_high > 0.5:
                            edge1 = another  # An edge attached to node_in
                            edge2 = an_edge  # An edge attached to node_out
                        else:
                            edge1 = an_edge
                            edge2 = another

                        # **First, the former node_in side:
                        new_pos = pt_in + t_low*dir_vec
                        new_d_node = Node('dmg',new_pos,np_dmg)

                        # change the length
                        if edge1.prm.has_key('l'):
                            edge1.prm['l'] = t_low*edge1.prm['l']
```

```
else:
    edge1.prm['l'] = self.dflt_prm['l']
if edge1.prm.has_key('r') == False:
    edge1.prm['r'] = self.dflt_prm['r']

# if there is more 50% of damage, the edge is
# assumed to be just a pipe (default component
# model), which doesn't have any control device.
if t_low < 0.5:
    edge1.model=self.dflt_model
    edge1.cv = {}
edge1.node_out = new_d_node
node['dmg'].append(new_d_node)
if edge1 == an_edge:
    edge[edge1.type].remove(edge1)
edge1.type = 'dmg'
edge1.get_pos()
edge['dmg'].append(edge1)

# **Second, the former node_out side:
new_pos = pt_in + t_high*dir_vec
new_d_node = Node('dmg',new_pos,np_dmg)
edge2.node_in = new_d_node

# change the length
if edge2.prm.has_key('l'):
    edge2.prm['l'] = (1-t_high)*edge2.prm['l']
else:
    edge2.prm['l'] = self.dflt_prm['l']
if edge2.prm.has_key('r') == False:
    edge2.prm['r'] = self.dflt_prm['r']

```

```
                        # if there is more 50% of damage, the edge is
                        # assumed to be just a pipe (default component
                        # model), which doesn't have any control device.
                        if t_high >= 0.5:
                            edge2.model=self.dflt_model
                            edge2.cv = {}
                        node['dmg'].append(new_d_node)
                        if edge2 == an_edge:
                            edge[edge2.type].remove(edge2)
                        edge2.type = 'dmg'
                        edge2.get_pos()
                        edge['dmg'].append(edge2)

                    elif (t_low > 0 and t_low <= 1) and t_high >= 1:
                        # ****node_out is damaged

                        # change the node to DamageNode
                        new_pos = pt_in + t_low*dir_vec
                        new_d_node = Node('dmg',new_pos,np_dmg)
                        an_edge.node_out = new_d_node

                        # change the length
                        if an_edge.prm.has_key('l'):
                            an_edge.prm['l'] = t_low*an_edge.prm['l']
                        else:
                            an_edge.prm['l'] = self.dflt_prm['l']
                        if an_edge.prm.has_key('r') == False:
                            an_edge.prm['r'] = self.dflt_prm['r']

                        # if there is more 50% of damage, the edge is
                        # assumed to be just a pipe (default component
                        # model), which doesn't have any control device.
```

```
                if t_low <= 0.5:
                    an_edge.model=self.dflt_model
                    an_edge.cv = {}
                if node_out in node:
                    node[node_out.type].remove(node_out)
                node['dmg'].append(new_d_node)
                edge[an_edge.type].remove(an_edge)
                an_edge.type = 'dmg'
                an_edge.get_pos()
                edge['dmg'].append(an_edge)

        elif an_edge.type == 'src' or an_edge.type =='snk':
            # In the case of source or sink edges, do the following.

            if t_low > 1 or t_high < 0:
                pass
            elif t_low <= 0:
                # ***node_in is damaged

                # create a DamageNode
                if node_in in node[node_in.type]:# Remove node-in
                    node[node_in.type].remove(node_in)

                if t_high < 1:
                    new_pos = pt_in + t_high*dir_vec
                    new_d_node = Node('dmg',new_pos,np_dmg)

                    # If it's source, use default model
                    if an_edge.type == 'src':
                        an_edge.model=self.dflt_model

                    # change the length
                    if an_edge.prm.has_key('l'):
                        an_edge.prm['l'] \
                        = (1-t_high)*an_edge.prm['l']
                    else:
                        an_edge.prm['l'] = self.dflt_prm['l']
                    if an_edge.prm.has_key('r') == False:
                        an_edge.prm['r'] = self.dflt_prm['r']
                    an_edge.node_in = new_d_node
                    node['dmg'].append(new_d_node)
                    edge[an_edge.type].remove(an_edge)
                    an_edge.type = 'dmg'
                    an_edge.get_pos()
                    edge['dmg'].append(an_edge)

                elif t_high >= 1: # Entire edge is destroyed.
                    if node_out in node[node_out.type]:
                        node[node_out.type].remove(node_out)
                    edge[an_edge.type].remove(an_edge)

            elif t_low > 0:
                # ****node_out is damaged

                # change the node to DamageNode
                if t_high < 1: # Only inner body damaged
                    an_edge2 = copy.deepcopy(an_edge)
                    an_edge2.node_out = an_edge.node_out
                    new_pos = pt_in + t_high*dir_vec
                    new_d_node = Node('dmg',new_pos,np_dmg)
                    an_edge2.node_in = new_d_node
                    if an_edge2.type == 'src':
                        an_edge2.model=self.dflt_model

```

```
    # change the length
    if an_edge2.prm.has_key('l'):
        an_edge2.prm['l'] = t_high*an_edge.prm['l']
    else:
        an_edge2.prm['l'] = self.dflt_prm['l']
    if an_edge2.prm.has_key('r') == False:
        an_edge2.prm['r'] = self.dflt_prm['r']
    node['dmg'].append(new_d_node)
    an_edge2.type = 'dmg'
    an_edge2.get_pos()
    edge['dmg'].append(an_edge2)

elif t_high >= 1:
    if node_out in node:
        node[node_out.type].remove(node_out)

new_pos = pt_in + t_low*dir_vec
new_d_node = Node('dmg',new_pos,np_dmg)
an_edge.node_out = new_d_node

# if it's sink, use the default model:
if an_edge.type == 'snk':
    an_edge.model=self.dflt_model

# change the length
if an_edge.prm.has_key('l'):
    an_edge.prm['l'] = t_low*an_edge.prm['l']
else:
    an_edge.prm['l'] = self.dflt_prm['l']
if an_edge.prm.has_key('r') == False:
    an_edge.prm['r'] = self.dflt_prm['r']

```

```
                        node['dmg'].append(new_d_node)
                        edge[an_edge.type].remove(an_edge)
                        an_edge.type = 'dmg'
                        an_edge.get_pos()
                        edge['dmg'].append(an_edge)

        edge = g_model.edge
        for k in edge.keys():
            a_group = edge[k]
            for i in xrange(len(a_group)):
                a_group[i].id = i
        node = g_model.node
        for k in node.keys():
            a_group = node[k]
            for i in xrange(len(a_group)):
                a_group[i].id = i
        g_model.update_model()
        self.dmg_applied = True


class Solver:
    '''
    class doc
    '''
    def __init__(self,g_model,dt=0.05,abs_tol=1e-6,dp=10):
        # g_model: graph model object reference

        self.set_model(g_model)
        self.abs_tol = abs_tol      # Tolerance of the solver
        self.dp = dp                # dP in the Newton solver
        self.dt = dt                # Sim time step
        self.recorder = None        # Object reference of data recorder
        self.dmg_bubble = None        # Object reference of damage bubble
        self._dmg_applied = False     # Flag. If damage is applied, True.
        self._conv_flag = 1           # Convergence flag

    #~ def __call__(self):


    def set_model(self,g_model):
        # Initialize solve according to graph model.

        self._model = g_model
        n_nodes = len(g_model.node['nrml'])
        self._np = zeros(n_nodes)
        g_model.J = zeros((n_nodes,n_nodes))
        self.fail = 0


    def run_to_next_time_step(self):
        # Run simulation for one time step.

        g_model = self._model
        dmg_bbl = self.dmg_bubble
        recorder = self.recorder

        # Apply damage when sim_t >= t_dmg
        if (dmg_bbl is not None) and \
           (dmg_bbl.dmg_applied is False) and \
           (g_model.t >= dmg_bbl.t_dmg):
            if recorder != None:
                recorder.stack_model(g_model)
            dmg_bbl(g_model)
            g_model.empty_data()

        nodes = g_model.node
        edges = g_model.edge
        np = g_model.get_np()
        abs_tol = self.abs_tol

        # Store all the data to local storages.
        [a_node.stack_res() for a_node in \
        nodes['nrml']+nodes['ref']+nodes['dmg']]
        edge_list = edges['nrml']+edges['src']+edges['snk']+edges['dmg']
        [an_edge.stack_res() for an_edge in edge_list]
        g_model.t_dat.append(g_model.t)

        cnt = 0
        self._conv_flag = 1 # reset _conv_flag to 1
        while cnt <= 150:
            # Iterative solver routine

            # Update x(t).
            [an_edge.get_flow() for an_edge in edge_list]
            g_model.setq()

            # Compute KCL equation. Stop iteration if KCL is satified.
            kcl_eq = inner(g_model.A,g_model.q)
            if norm(kcl_eq) <= abs_tol:
                break
            self.jacobian()      # Compute Jacobian

            # Update the node pressure.
            d_np = solve(g_model.J, kcl_eq)
            max_mv = max(abs(d_np))
            if max_mv > 20000:
                # This is just for improving numerical stability.
                # If the dP estimates are too large, just shrink them.
                np = np - (20000/max_mv)*d_np
            else:
                np = np - d_np
            g_model.set_np(np)
            cnt+=1

        # Update x(t-1) for next time step. x(t-1)=x(t).
        for an_edge in edge_list:
            an_edge.xt_1 = an_edge.xt
        g_model.t += self.dt

        # If simulation have difficulties in convergence, give warning
        # message.
        if cnt >= 100:
            e = norm(kcl_eq)
            print "Solution␣not␣converged␣at␣t="+str(g_model.t)+",␣with␣e␣="\
              +str(e)
            if e > 8e-6:
                # Worst case, flag down, so controllers skip the outputs.
                print"convergence␣flag␣down"
                self._conv_flag = 0



    def jacobian(self):
        # jacobian J from KCL equation set
        # See Algorithm 2.
        dp = self.dp
        g_model=self._model
        A = g_model.A
        L= g_model.L
        edges = g_model.edge['nrml']+g_model.edge['src'] \
                +g_model.edge['snk']+g_model.edge['dmg']
        n_nodes,n_edges = A.shape
        J = zeros((n_nodes,n_nodes))
        dqdp_vec = zeros(n_edges)
        dqdp_vec = [(an_edge(bc=an_edge.ep+dp) - an_edge.xt)/dp \
                    for an_edge in edges]
        for i in xrange(n_nodes):
            nonzero_in_Ai = self._find(A[i],0,False)
            nonzero_in_Li = self._find(L[i],0,False)
            for j in nonzero_in_Li:
                if j == i:
                    sum = 0
                    for k in nonzero_in_Ai:
                        sum += dqdp_vec[k]
                    J[i,j]=sum
                else:
                    for k in nonzero_in_Ai:
                        if A[j,k] != 0:
                            J[i,j] = -dqdp_vec[k]
                            break
        g_model.J=J

    def _find(self,vec,value,logic=True):
        y = []
        for i in xrange(len(vec)):
            if logic is True:
                if vec[i] == value:
                    y.append(i)
            else:
                if vec[i] != value:
                    y.append(i)
        y = array(y)
        return y

    model = property(fget=lambda self: self._model,fset=set_model)
```

B.3.4 control.py

```
from numpy import *
from numpy.linalg import norm

def edge_adj_mat(A):
    # Create an edge adjacency matrix
    adj_mat = inner(A.T,A.T)-2*eye(A.shape[1])
    return adj_mat


def sv_inc(edge_id, A):
    # Create an incidence matrix of the controller-attached edges.
    #   inputs:
    #       edge_id = indices of the controller-attached edges.
    #             A = Incidence matrix of graph model.

    id_set = copy(edge_id)

    # Next, perform node merging
    A_sm = copy(A)
    edges = range(A.shape[1])
    new_list = []
    for e in edges:
        if e not in edge_id:
            new_list.append(e)
    edges = new_list
    for e in edges:
```

```python
        row = []
        col = A_sm[:,e]
        for i,v in enumerate(col):
            if v != 0:
                row.append((i,A_sm[i,:]))
            if len(row) == 2:
                break
        if row==[]:
            pass
        else:
            A_sm[row[0][0]] = row[0][1] + row[1][1]
            A_sm = delete(A_sm, row[1][0],0)
    A_sm = delete(A_sm,edges,1)
    # M = edge_adj_mat(A_sm)
    return A_sm

class OPin:
    # Represent a data signal pin.
    def __init__(self):
        self.q = None        # Flow rate
        self.hs = None       # Health status
        self.dmg = False     # Damage flag


class Port:
    # Port that contains two pins, input- and output- pins.
    def __init__(self,an_agent=None):
        self.o_pin = OPin()          # Output-pin
        self.to = None               # Controller object reference
        self.i_pin = None            # Input-pin
        if an_agent != None:
            self.to = an_agent
```

```



class Agent:
    # Base controller class definition.
    def __init__(self):
        self.ports_nb = []      # List of ports for neighbors
        self.ports_hl = []      # List of ports for upper-layer units
        self.ports_ll = []      # List of ports for Lower-layer units
        self.dt = 0             # Controller time step
        self.health = 1         # Controller initial health status

    def update_self(self):
        # Update internal states.

        pass

    def process(self):
        # Control law definitions.

        pass

    def update_ports(self):
        # read and write the values to the ports
        pass



class SmartValve(Agent):
    # Smart valve controllers on pipelines.

    def __init__(self,an_edge,dt = 0.05, valve_id = 'v1'):
        Agent.__init__(self)
        # Connection with a lower unit
        self.ports_ll = an_edge # An edge with valve control variable
        self.dt = dt
        self._v_id = valve_id   # Valve name in cv dictionary in an edge.
        self._dmg = False
        self.valve_speed = 0.25 # Valve open and close speed. /sec.
        self.adj_mat = [[],[]]  # Local adjacency matrix


    def update_self(self):
        an_edge = self.ports_ll
        self.q = an_edge.xt_1
        self.np = (an_edge.node_in.np + an_edge.node_out.np)/2
        self.health = self.health_sig()
        if self.health != 0:
            self.v = an_edge.cv['v1']


    def process(self):
##        self.update_self()
        if self.health != 0:
            if self._dmg == False:
                if self.dmg_test() == True:
                    self._dmg = True
                else:
                    for a_port in self.ports_nb:
                        if a_port.i_pin.hs == 0:
                            self._dmg = True
                            break
            elif self._dmg == True and self.v > 0:
                if self.v > 0.1:
                    self.v = self.v - self.valve_speed*self.dt
                else:
                    self.v = self.v - 0.5*self.valve_speed*self.dt

                    if self.v < 0:
                        self.v = 0
            # Update ctrls of its plant:
            self.ports_ll.cv['v1'] = self.v

    def update_ports(self):
        for a_port in self.ports_nb:
            a_port.o_pin.q = self.q
            a_port.o_pin.hs = self.health



    def dmg_test(self):
        # Check local flow conservation
        A = array(self.adj_mat)
        Q = array([x.i_pin.q for x in self.ports_nb])
        fc = inner(A,Q)+array([self.q,-self.q])
##        if fc[0] < -1e-5 or fc[1] < -1e-5:
        if norm(fc) >= 1.5e-5:
        # Check if ther is net sink between itself and neighbors.
            return True
        else:
            return False


    def health_sig(self):
        if self.ports_ll.cv == {}:
            return 0
        else:
            return 1


class CpAgent(Agent):
    # Controller unit for chiller-pump subnetwork

    def __init__(self, dt = 0.05, op_ps = 200):
        Agent.__init__(self)
        self.port_pmp = []
        self.dt = dt
        self._dmg = False
        self.valve_speed = 0.25
        # Pump speed rise per sec when turned on.
        # When turned off, pump_speed*1.5.
        self.pump_speed = 50
        self.op_pump_speed = op_ps
        self.adj_mat = [[],[]]



    def update_self(self):
        lu = self.ports_ll
        self.q = [an_edge.xt_1 for an_edge in lu]
        self.health = self.health_sig()
        if self.health != 0:
            self.v = array([an_edge.cv['v1'] for an_edge in\
                [lu[0],lu[2],lu[3]] ])
            self.ps = self.ports_ll[2].cv['ps1']

    def process(self):
        #~ if self.health != 0:
```

```
        if self._dmg == False:
            if self.dmg_test() == True:
                # Check the damage of itself
                self._dmg = True
            else:
                # Check if neighbors' health
                for a_port in self.ports_nb:
                    if a_port.i_pin.hs == 0:
                        self._dmg = True
                        break
                # If neighboring pump net is damaged, turn itself on:
                if self.port_pmp.i_pin.dmg == True:
                    if self.v[0] < 1:
                        self.v = self.v + self.valve_speed*self.dt*ones(3)
                        if self.v[0] > 1:
                            self.v = ones(3)
                    op_ps = self.op_pump_speed
                    if self.ps < op_ps:
                        self.ps = self.ps + self.pump_speed*self.dt
                        if self.ps > op_ps:
                            self.ps = op_ps
                    # Update ctrls to its plant:
                    self.ports_ll[0].cv['v1']=self.v[0]
                    self.ports_ll[3].cv['v1']=self.v[2]
                    self.ports_ll[2].cv['ps1'] = self.ps

        elif self._dmg == True:
            if self.v[0] > 0.1:
                self.v = self.v - self.valve_speed*self.dt*ones(3)
            elif self.v[0] > 0:
                self.v = self.v - 0.5*self.valve_speed*self.dt*ones(3)
                if self.v[0] < 0:
                    self.v = array([0,0,0])
            if self.ps > 0:
                self.ps = self.ps - self.pump_speed*self.dt
                if self.ps < 0:
                    self.ps = 0
            # Update ctrls to its plant:
            self.ports_ll[2].cv['ps1'] = self.ps
            self.ports_ll[0].cv['v1']=self.v[0]
            self.ports_ll[3].cv['v1']=self.v[2]
            #~ self.ports_ll[2].cv['v1']=self.v[1]

    def update_ports(self):
        adj_mat = self.adj_mat
        # in-node connections:
        for i,v in enumerate(adj_mat[0]):
            if v != 0:
                self.ports_nb[i].o_pin.q = self.q[1]
                self.ports_nb[i].o_pin.hs = self.health
        # out-node connections:
        for i,v in enumerate(adj_mat[1]):
            if v != 0:
                self.ports_nb[i].o_pin.q = self.q[2]
                self.ports_nb[i].o_pin.hs = self.health
        # pump net communication:
        self.port_pmp.o_pin.dmg = self._dmg
        self.port_pmp.o_pin.hs = self.health



    def dmg_test(self):
        # Check local flow conservation
        A = array(self.adj_mat)
        Q = array([x.i_pin.q for x in self.ports_nb])
        fc = inner(A,Q)+array([self.q[1],-self.q[2]])
        if norm(fc) >= 1.5e-5:
            # Check if ther is net sink between itself and neighbors.
            return True
        else:
            return False


    def health_sig(self):
        lu = self.ports_ll
        hs = 1
        for i in [0,2,3]:
            if lu[i].cv == {}:
                hs = 0
                self._dmg = True
                break
        return hs


class CtrlSystem:
    # Base control system class
    def __init__(self):
        self.agent = []
        self._edge_id = []
        self.n_layer = 0


class DmgCtrlSys(CtrlSystem):
    # Damage control system class
    def __init__(self):
        CtrlSystem.__init__(self)

    def setup_cp_agents(self, g_model, dt=0.05, key='pc', n_cp=2):
        # Scan the edges in graph model and
        # create chiller-pump controller objects

        edges = g_model.edge['nrml']+g_model.edge['src']+g_model.edge['snk']
        for cp_id in range(1,n_cp+1):
            cp = CpAgent(dt)
            cp.ports_ll = [None]*4
            e_id = [0]*4
            for i,an_edge in enumerate(edges):
                if ('in_'+key+str(cp_id)) in an_edge.name:
                    cp.ports_ll[0] = an_edge
                    e_id[0] = i

                elif (key+str(cp_id)+'_in') in an_edge.name:
                    cp.ports_ll[1] = an_edge
                    e_id[1] = i

                elif (key+str(cp_id)+'_out') in an_edge.name:
                    cp.ports_ll[2] = an_edge
                    e_id[2] = i

                elif ('out_'+key+str(cp_id)) in an_edge.name:
                    cp.ports_ll[3] = an_edge
                    e_id[3] = i

            if (self.agent != []) and (self._edge_id[-1] > e_id[-1]):
                for i,x in enumerate(self._edge_id):
                    if e_id[-1] < x[1]:
                        self.agent.insert(i,cp)
                        self._edge_id.insert(i,e_id[-1])
            else:
                self.agent += [cp]
                self._edge_id += [e_id[-1]]

    def setup_smart_valves(self, g_model,dt):
        # Scan the edges in graph model, and create smart valve objects.

        edges = g_model.edge['nrml']
        for m,an_edge in enumerate(edges):
            if ('pipe' in an_edge.name) or ('bps' in an_edge.name):
                if 'v1' in an_edge.cv.keys():
                    id = m
                    sv = SmartValve(an_edge,dt,'v1')
                    if (self.agent != []) and (self._edge_id[-1] > id):
                        for i,x in enumerate(self._edge_id):
                            if id < x:
                                self.agent.insert(i,sv)
                                self._edge_id.insert(i,id)
                                break
                    else:
                        self.agent += [sv]
                        self._edge_id += [id]


    def connect(self,g_model):
        # Connect all the controller objects through their ports
        # according to the incidence matrix of controller-attached
        # edges and the edge adjacency matrix

        ag = self.agent
        edge_id = self._edge_id
        n_ag = len(ag)
        # Connect all the smart valves and pump network ctrlers:
        A_ag = sv_inc(edge_id, g_model.A_tot)
        M = edge_adj_mat(A_ag)
        for i, an_ag in enumerate(ag):
            for j,a in enumerate(A_ag[:,i]):
                if a == 1:
                    row1 = A_ag[j,:]
                elif a == -1:
                    row2 = A_ag[j,:]
            for j in xrange(i,n_ag):
                if M[i,j] != 0:
                    an_ag.ports_nb += [Port(ag[j])]
                    ag[j].ports_nb += [Port(an_ag)]
                    an_ag.ports_nb[-1].i_pin = ag[j].ports_nb[-1].o_pin
                    ag[j].ports_nb[-1].i_pin = an_ag.ports_nb[-1].o_pin
            for j,a in enumerate(M[i,:]):
                if j != i and a != 0:
                    an_ag.adj_mat[0] += [row1[j]]
                    an_ag.adj_mat[1] += [row2[j]]
        # Connect among pump network ctrlers:
        pmp_ag = []
        for an_ag in ag:
            if isinstance(an_ag,CpAgent):
                pmp_ag += [an_ag]
        n = len(pmp_ag)
        for i in xrange(n-1):
            pmp_ag[i].port_pmp = Port(pmp_ag[i+1])
            pmp_ag[i+1].port_pmp = Port(pmp_ag[i])
            pmp_ag[i].port_pmp.i_pin = pmp_ag[i+1].port_pmp.o_pin
            pmp_ag[i+1].port_pmp.i_pin = pmp_ag[i].port_pmp.o_pin
```

B.3.5 post_proc.py

```
from copy import deepcopy
```

```python
from pylab import *
import cen_model

class Recorder:
    # Records simulation result.
    # Used to plot and retrieve the result.

    def __init__(self):
        self.storage = []   # Storage of multiple simulation results.
        self.size = 0       # Size of storage
        self.res = []       # Result of a single simulation run
        self._g_model = None

    def stack_model(self,g_model):
        # Stores a graph model into the result. all the data is stored locally
        # inside graph model.

        self.res += [deepcopy(g_model)]

    def stack_res(self):
        # Stacks the result into storage.

        self.storage += [self.res]
        self.size += 1
        self.res = []
        self._g_model = None

    def get_res(self,res_id):
        # Retrieve a result by the simulation no.

        return self.storage[res_id]

    def plot(self, res_id, edge_name, var = None):
        # Plot a result of stored in a specific edge.
        # Inputs:
        #       res_id = result, or simulation no.
        #       edge_name = name of an edge
        #       var = name of the variable to be investigated

        res = self.get_res(res_id)
        edge_name = [edge_name]

        dat_set = {}
        for a_name in edge_name:
            dat_set[a_name] = None
        for i,mdl in enumerate(res):
            edges = mdl.edge['nrml']+mdl.edge['src']\
                   +mdl.edge['snk']+mdl.edge['dmg']
            for an_edge in edges:
                if an_edge.name in edge_name:
                    a_name = an_edge.name
                    if dat_set[a_name] == None:
                        dat_set[a_name] = an_edge.res_dat.copy()
                        dat_set[a_name]['in_node'] = \
                        an_edge.node_in.res_dat[:]
                        dat_set[a_name]['out_node'] = \
                        an_edge.node_out.res_dat[:]
                    else:
                        for k,v in an_edge.res_dat.iteritems():
                            dat_set[a_name][k] += v
                        dat_set[a_name]['t'] += mdl.t_dat[:]
                        dat_set[a_name]['in_node'] += \
                        an_edge.node_in.res_dat[:]
                        dat_set[a_name]['out_node'] += \
                        an_edge.node_out.res_dat[:]
            if i == 0:
                dat_set[a_name]['t'] = mdl.t_dat[:]
            else:
                dat_set[a_name]['t'] += mdl.t_dat[:]
        figure()

        if var == None:
            # Plot xt_1 first
            subplot(211)
            hold(True)
            for a_name in dat_set.iterkeys():
                dat = dat_set[a_name]
                n = len(dat['xt_1'])
                t = dat['t'][:n]
                plot(t,dat['xt_1'],label=a_name)
            hold(False)
            title('Result No. '+str(res_id))
            ylabel('Vol. flow rate (m^3/s)')
            xlabel('Time (sec)')
            legend()
            grid(True)
            # Plot np
            subplot(212)
            hold(True)
            for a_name, dat in dat_set.iteritems():
                dat = dat_set[a_name]
                n = len(dat['in_node'])
                t = dat['t'][:n]
                plot(t,dat['in_node'],label=a_name+', in_node')
                n = len(dat['out_node'])
                t = dat['t'][:n]
                plot(t,dat['out_node'],label=a_name+', out_node')
            hold(False)
            ylabel('Presure (Pa)')
            xlabel('Time (sec)')
            legend()
            grid(True)
            subplot(212)
            hold(True)
        else:
            hold(True)
            for a_name in dat_set.iterkeys():
                dat = dat_set[a_name]
                n = len(dat[var])
                t = dat['t'][:n]
                plot(t,dat[var],label=a_name)
            hold(False)
            title('Result No.'+str(res_id))
            ylabel(var)
            xlabel('Time (sec)')
            legend()
            grid(True)
        show()

    def op_capa_rate(self,o_mode=0):
        '''
        op_capa_rate(self,o_mode=0):
            o_mode = 0 : simple plotting
                   = 1 : return the tuple (expt_no,op_capa_rate) of list
                   = 2 : Do both
        Compute the operation capability rates of all the simulation runs
        stored.
        '''
        def op_cap(edge, idx = 0):
            '''
            idx : for the baseline capacity estimation, 0
                  for final status, -1
            Edit 'e' and 'w' if you want model and weight for
            different mission scheme
            '''
            # Name of the edges with service loads:
            #     IEP,   Eng_rm, Radar,  CIC1,    CIC2,    FCell,  EMgun
            e = ['svc1','svc2','svc3', 'svc4',   'svc5',   'svc6', 'svc7']
            # Weights of each loads, 1 to 10
            #    IEP,Eng_rm,Radar, CIC,    FCell, EMgun
            w =  [2,    6,     5,      8,      4,      6]

            y = []
            if idx == 0:
                for x in e:
                    y += [abs(edge[x].res_dat['xt_1'][idx])]
                op_cap.y0 = y
            else:
                for i,x in enumerate(e):
                    y_temp = abs(edge[x].res_dat['xt_1'][idx])
                    if y_temp > op_cap.y0[i]:
                        y_temp = op_cap.y0[i]
                    y += [y_temp]
            C = w[0]*y[0] + w[1]*y[1] + w[2]*y[2]
            if y[3]>=y[4]:
                C += w[3]*y[3]
            else:
                C += w[3]*y[4]
            C += (w[4]*y[5] + w[5]*y[6])
            return C

        n = len(self.storage)
        op_c_dat = []
        for i in xrange(n):
            res = self.get_res(i)
            edge = res[0].get_edge_dict()
            t_dmg = res[0].t_dat[-1]

            # Compute initial capability:
            C0 = op_cap(edge,0)
            for j,t in enumerate(res[1].t_dat):
                if t > t_dmg+8:
                    cr_t = j
                    break

            # Check if the system went total failure by the damage:
            # If the rupture is not isolated for 8 secs after rupture,
            # the system is considered as total failure.
            tot_q = 0
            for an_edge in res[1].edge['dmg']:
                if isinstance(an_edge, cen_model.CoupledEdge):
                    pass
                else:
                    tot_q += an_edge.res_dat['xt_1'][cr_t]
            if tot_q > 2e-5:
                tot_failure = True
            else:
                tot_failure = False

            # Compute the final capability after damage:
            if tot_failure == False:
                edge = res[1].get_edge_dict()
                C = op_cap(edge,-1)
            else:
                C = 0
            op_c_dat += [C/C0]

        if o_mode==0 or o_mode==2:
            figure()
            plot(range(n),op_c_dat,'-o')
            x = []
            for i in range(n):
                x += [str(i)]
            xticks(range(n),x)
            xlabel("Simulation␣no.")
            ylabel("Operation␣capability␣rate")
            title("Operation␣capability␣comparison")
            xlim(0,range(n))
            ylim(0,1)
            grid(True)
            show()
        if o_mode==1 or o_mode==2:
            return (range(n),op_c_dat)

    def whos_down(self,res_id,o_mode=0):
        '''
        whos_down(self,res_id,o_mode=0):
            res_id   :result id
            o_mode   :=0, print the list
                      =1, return the list

        Function for checking which service components are down.
        '''
        mdl=self.get_res(res_id)[-1]
        down_list = []
        for k,e in mdl.get_edge_dict().iteritems():
            if 'svc' in k:
                q = e.xt_1
                if abs(q) < 1e-5:
                    print k+" (",q,")"
                    down_list.append(k)
            if 'pc' in k:
                if e.cv.has_key('ps1'):
                    ps1=e.cv['ps1']
                    if ps1 < 10:
                        print k+" is off or down (", ps1, " rad/s)"
                        down_list.append(k)
                    else:
                        print k+" is on (", ps1, " rad/s)"


    def get_edge_dat(self,res_id,edge_name,o_mode=0,f_name=None):
        '''
        get_edge_dat(res_id,edge_name,o_mode=0):
            res_id    : result id
            edge_name : a single string of an edge name
            o_mode    : = 0, returned output
                        = 1, .xls file, default name is the edge name
        Just provides the data of a variable, instead of plotting.
        '''
        res = self.get_res(res_id)
        dat = {'t':[],'p_in':[],'p_out':[]}
        for mdl in res:
            e = mdl.get_edge_dict()
            a_e = e[edge_name]

            for k,d in a_e.res_dat.iteritems():
                if dat.has_key(k):
                    dat[k]+=d
                else:
                    dat[k]=d
            dat['p_in']+=a_e.node_in.res_dat
            dat['p_out']+=a_e.node_out.res_dat[:]
            dat['t']+=mdl.t_dat[:]

        if o_mode == 1:
            import pyExcelerator as pyx
            wb = pyx.Workbook()
            ws = wb.add_sheet('0')
            col= 0
            ws.write(0,col,'t')
            for i,v in enumerate(dat['t']):
                ws.write(i+1,col,str(v))
            col=1
            for k,d in dat.iteritems():
                if k == 't':
                    pass
                else:
                    ws.write(0,col,k)
                    for i,v in enumerate(d):
                        ws.write(i+1,col,str(v))
                    col+=1

            if f_name == None:
                f_n = "res_" + str(res_id) + "_" + edge_name + ".xls"
            else:
                f_n = f_name+".xls"
            wb.save(f_n)
        else:
            return dat


    def save_res_dat(self,f_name):
        # Save the current data
        # File extension is ”.res.”

        import cPickle as p
        f = open(f_name+”.res”,’w’)
        p.dump(self.storage,f)

    def load_res_dat(self,f_name):
        # Load a saved data
        # File extension is ”.res.”

        import cPickle as p
        f = open(f_name+”.res”,’r’)
        self.storage=p.load(f)
        self.size=len(self.storage)

def check_var(mdl,var_name):
    # Check final values of a certain variable or parameter in every edge
    # of a graph model
    #   input:
    #       var_name = name (string) of a variable or parameter

    edges=mdl.edge[’nrml’]+mdl.edge[’dmg’]+mdl.edge[’src’]+mdl.edge[’snk’]
    for e in edges:
        if e.cv.has_key(var_name):
            print e.name,”:”, e.cv[var_name]
        elif e.prm.has_key(var_name):
            print e.name,”:”, e.prm[var_name]
        elif var_name == ”xt_1”:
            print e.name,”:”, e.xt_1
        elif var_name == ”xt”:
            print e.name,”:”, e.xt
        elif var_name == ”ep”:
            print e.name,”:”, e.ep
        elif var_name == ”np”:
            print e.name,”:”, e.node_in.np,”,”, e.node_out.np
```

REFERENCES

[1] "About Bond Graphs." [Online] http://www.bondgraph.info/about.html. [Retrieved April 8, 2008].

[2] "Britannica Online Encyclopedia." [Online] http://www.britannica.com. [Retrieved April 16, 2008].

[3] "DC (Damage Control) Museum, NAVSEA-USS Stark." [Online] http://www.dcfp.navy.mil/mc/museum/STARK/Stark3.htm. [Retrieved February 25, 2008].

[4] "USS Princeton (CG 59)." [Online] http://www.navybuddies.com/cg/cg59.html. [Retrieved February 26, 2008].

[5] "USS Stark on Fire." [Online] http://www.navybook.com/nohigherhonor/pic-stark.shtml. [Retrieved February 25, 2008].

[6] "Yard patrol craft (yp 676)." [Online] http://www.boats.dt.navy.mil/pg2/YP676.htm. [Retrieved November 06, 2008].

[7] *NIST/SEMATECH e-Handbook of Statistical Methods*. National Institute of Standards and Technology, October 2008. http://www.itl.nist.gov/div898/handbook/.

[8] Aadaleesan, P., Miglan, N., Sharma, R., and Saha, P., "Nonlinear System Identification Using Wiener Type Laguerre – Wavelet Network Model," *Chemical Engineering Science*, vol. 63, pp. 3932–3941, April 2008.

[9] Al-Duwaish, H., Karim, M., and Chandrasekar, V., "Use of multilayer feedforward neural networks in identification and control of wiener model," in *IEEE Proceedings-Control Theory and Applications*, vol. 143, pp. 255–258, 1996.

[10] Bachkosky, J., D. Katz, R. R., and Weldon, W., "Naval Electromagnetic (EM) Gun Technology Assessment," Tech. Rep. NRAC 04-01, Naval Research Advisory Committee, 800 North Quincy Street Arlington, VA 22217-5660, February 2004.

[11] Bai, E.-W., "Decoupling the linear and nonlinear parts in hammerstein model identification," *Automatica*, vol. 40, pp. 671–676, 2004.

[12] Baltersee, J. and Chambers, J., "Nonlinear adaptive prediction of speech with a pipelined recurrent neural network," *IEEE Transactions on Signal Processing*, vol. 46, pp. 2207–2216, August 1998.

[13] Bellman, R. E., *Adaptive Control Processes: A Guided Tour*. New York: Princeton University Press, 1961.

[14] Bollobás, B., *Modern Graph Theory*. Graduate Texts in Mathematics, New York, NY: Springer, 1998.

[15] Croegaert, M., Shapiro, S., Callahan, B., and Roach, J., "Evaluating Intelligent Fluid Automation Systems using a Fluid Network Simulation Environment," in *13th International Ship Control Systems Symposium*, (Orlando, Florida), 2003.

[16] Davis, T. W. and Palmer, R. W., *Computer-Aided Analysis of Electrical Networks*. Columbus, Ohio: Charles E. Merrill Publishing Co., 1973.

[17] Deo, M., *Graph Theory with Applications to Engineering and Computer Science*. Series in Automatic Computation, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1974.

[18] Diestel, R., *Graph Theory.* Graduate Texts in Mathematics, Heidelberg, NY: Springer, 3rd ed., 2005.

[19] Doerry, N., Robey, H., Amy, J., and Petry, C., "Powering the Future with the Integrated Power System," *Naval Engineering Journal*, pp. 267–282, May 1996.

[20] Du, K. L. and Swamy, M., *Neural Networks in a Softcomputing Framework.* Springer, 2006.

[21] Dunnington, L., Stevens, H., and Grater, G., "Integrated Engineering Plant for Future Naval Combatants - Technology Assessment and Demonstration Roadmap," Tech. Rep. MSD-50-TR-2003/01, Anteon Corp., January 2003.

[22] Etman, L., "Design and Analysis of Computer Experiments: the method of Sacks et al.," in *Engineering Mechanics Report*, no. WFW 94.098, Eindhoven University of Technology, 1994.

[23] Evans, J. R. and Minieka, E., *Optimization Algorithm for Networks and Graphs.* New York: Mercel Dekker Inc., 2nd ed., 1992.

[24] Fahlman, S. E. and Lebiere, C., "The cascade-correlation learning architecture," Tech. Rep. CMU-CS-90-100, School of Computer Science, Carnegie Mello University, Pittsburgh, PA 15213, 1991.

[25] Fritzson, P., *Tutorial, Introduction of Object-Oriented Modeling and Simulation with Open-Modelica*, 2006.

[26] Gevers, M., *System Identification without Lennart Ljung: What would have been different?*, ch. 13, pp. 61–85. Lund, Sweden: Studentlitteratur, 2006.

[27] Gilmore, M., "The Navy's DD(X) Destroyer Program." Congressional Budget Office testimony, July 2005.

[28] Giri, F., Chaoui, F., Haloua, M., Rochdi, Y., and Naitali, A., "Hammerstein model identification," in *Proceedings of the 10 th Mediterranean Conference on Control and Automation*, July 2002.

[29] Giunta, A., Watson, L., and Koehler, J., "A Comparison of Approximation Modeling Techniques: Polynomial Versus Interpolating Models," 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis & Optimization, September 1998.

[30] Gross, J. L. and Yellen, J., eds., *HandBook of Graph Theory.* Discrete Mathematics and Its Applications, Boca Raton, Florida: CRC Press LLC, 2004.

[31] Hagenblad, A., *Aspects of the Identification of Wiener Models.* PhD thesis, Linköpings University, SE-581 83 Linköping, Sweden, 1999.

[32] Hagenblad, A., Ljung, L., and Wills, A., "Maximum likelihood identification of wiener models," *Automatica*, vol. 44, pp. 2697–2705, 2008.

[33] Hecht-Nielson, R., "Kolmogorov's Mapping Neural Network Existence Theorem," in *IEEE International Conference on Neural Networks*, vol. 3, pp. 11–13, 1987.

[34] Janczak, A., *Identification of Nonlinear Systems Using Neural Networks and Polynomial Models: A Block-Oriented Approach.* Lecture Note in Control and Information Sciences, Berlin Heidelberg, Germany: Springer-Verlag, 2005.

[35] Jeong, S., Murayama, M., and Yamamoto, K., "Efficient optimization design method using kriging method," *Journal of Aircraft*, vol. 42, no. 2, pp. 413–420, 2005.

[36] Kocijan, J., Girard, A., Banko, B., and Murray-Smith, R., "Dynamic System Identification with Gaussian Process," *Mathematical and Computer Modelling of Dynamical Systems*, vol. 11, pp. 411–424, December 2005.

[37] Larock, B. E., Jeppson, R. W., and Watters, G. Z., *Hydraulics of Pipline Systems.* 2000 Corporate Blvd., N.W., Boca Raton, Florida 33431: CRC Press LLC, 2000.

[38] Lee, T. H. and Jung, J. J., "Kriging metamodel based optimization." School of Mechanical Engineering, Hanyang University, Seoul, South Korea, Accquired in 2006.

[39] Lively, K. A., Scheidt, D. H., and Drew, K. F., "Mission Based Engineering Plant Control," ASNE Rconfiguration and Survivability Symposium, February 2005.

[40] Ljung, L., "System identification, to be included in the control handbook, edited by w. levine," tech. rep., Dept of EE. Linköping University, S-581 83 Linköping, Sweden, May 1995. http://www.control.isy.liu.se/publications.

[41] Ljung, L., *System Identification.* PTR Prentice Hall Information and System Sciences Series, New Jersey: Prentice Hall, 1999.

[42] Ljung, L. and Glad, T., *Modeling of Dynamic Systems.* 113 Sylvan Avenue, Englewood Cliff, NJ 07632: Prentice Hall, Inc., 1994.

[43] Luers, A. C., Hil, S. A., Scheffey, J. L., Pham, H. V., and Farleyl, J. P., "The Evaluation of the Autonomic Fire Suppression System Concept of Operations and PDA Cooling Effectiveness," Tech. Rep. ADA417406, Naval Research Lab, Washington DC, September 2003.

[44] Mack, Y., Goel, T., Shyy, W., and Haftka, R., *Evolutionary Computation in Dynamic and Uncertain Environments*, vol. 51 of *Studies in Computational Intelligence.* Springer, 2007.

[45] Mandic, D. and Chambers, J., *Recurrent neural networks for prediction: learning algorithms, architectures, and stability.* New York: Wiley, 2001.

[46] Manik, A., Goplakrishnan, K., Singh, A., and Yan, S., "Neural Networks Surrogate Models for Simulatiing Payment in Pavement Construction," *Journal of Civil Engineering and Management*, vol. 14, no. 4, pp. 235–240, 2008.

[47] Margolis, D. and Shim, T., "Bond Graph Modelling for Non-Linear Hydro-Mechanical Systems," in *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, vol. 219, pp. 371–382, Professional Engineering Publishing, 2005.

[48] Mathworks^{TM}, *Simulink® 7 Getting Started Guide*, 2009.

[49] Mavris, D. N. and Kirby, M. R., "Technology identification, evaluation, and selection for commercial transport aircraft," in *58th Annual Conference of Society of Allied Weight Engineers, Inc*, (San Jose, California), Society of Allied Weight Engineers (SAWE), Inc., 24-26 May 1999.

[50] Meliopoulos, A. P. and Stefopoulos, G. K., "Improved Numerical Integration Method for Power/Power Electronic Systems Based on Three-Point Collocation," in *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference*, (Seville, Spain), pp. 6780–6787, December 2005.

[51] Modelica Association, "Modelica^{TM}— A Unified Object-Oriented Language for Physical Systems Modeling, Tutorial, Version 1.4." [Online] http://www.modelica.org/publications, December 2000.

[52] Moon, K., Weston, N., and Mavris, D., "A Method for Speeding Up the Time-Domain Simulation of a Complex System Using Surrogate Modeling Technique," in *ASNE Automation and Control Conference*, (Biloxi, MS), 2007.

[53] Myers, R. H. and Montgomery, D. C., *Response Surface Methodology:Process and Product Optimization Using Designed Experiments.* New York: John Wiley & Sons, Inc., 1995.

[54] Naval Surface Warfare Center Dahlgren Division, "Open Architecture (OA) Computing Environment Design Guidance, Version 1.0." Prepared for Program Executive Office, Integrated Warfare Systems, 17320 Dahlgren Road, Dahlgren VA 22448-5100, August 2004.

[55] NavSource Naval History, "USS Zumwalt (DDG-1000)." Online, http://www.navsource.org/archives/05/011000.htm, March 2010.

[56] Oden, J. T., Belytschko, T., Fish, J., Hughs, T. J., Johnson, C., Keyes, D., Laub, A., Petzold, L., Srolovitz, D., and Yip, S., "Simulation-based engineering and science: Revolutionizing engineering science through simulation." Report of the National Science Foundation Blue Ribbon Panel on Simulation-Based Engineering Science, May 2006.

[57] Ogunfunmi, T., *Adaptive Nonlinear System Identification: The Volterra and Wiener Model Approaches*. Signals and Communication Technology, Springer, 2007.

[58] O'Rourke, R., "Navy DDG-1000 Destroyer Program: Background, Oversight Issues, and Options for Congress." Congressional Research Service, Report for Congress, October 2007. RL32109.

[59] Pearson, R. K., *Discrete-Time Dynamic Models*. A Series of Textbooks and Monographs, 198 Medison Avenue, New York, New York 10016: Oxford University Press, Inc., 1999.

[60] Queipo, N. V., Haftka, R. T., Shyy, W., Goel, T., Vaidyanathan, R., and Tucker, K. P., "Surrogate-Based Analysis and Optimization," *Progress in Aerospace Sciences*, vol. 41, pp. 1–28, 2005.

[61] Rasmussen, C. E. and Williams, C., *Gaussian Processes for Machine Learning*. No. ISBN 0-262-18253-X, the MIT Press, 2006.

[62] Sacks, J., Welch, W. J., Mitchell, T. J., and Wynn, H. P., "Design and Analysis of Computer Experiments," *Statistics Science*, vol. 4, pp. 409–423, November 1989.

[63] Samarasinghe, S., *Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition*. Boca Raton, FL: Auerbach Publications, 2007.

[64] SAS Institute Inc., *JMP Design of Experiments Guide Release 7*. SAS Institute Inc., Cary, NC, 2007.

[65] Scharl, J. and Mavris, D., "Building Parametric and Probabilistic Dynamic Vehicle Models using Neural Networks," in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, (Montreal, Canada), August 2001.

[66] Scheidt, D. H., "Intelligent Agen-Based Control," in *Johns Hopkins APL Technical Digest*, vol. 23, pp. 383–395, 2002.

[67] Schetzen, M., *The Volterra and Wiener Theories of Nonlinear Systems*. John Wiley & Sons, Inc., 1980.

[68] Seiffert, U., "Training of Large-Scale Feed-Forward Neural Networks," International Joint Conference on Neural Networks, July 2006.

[69] Seman, A. J., Toomey, K., and Lang, S., "Reduced Ship's crew-by Virtual Presence (RSVP) Advanced Technology Demonstration (ATD) Final Report," Tech. Rep. NSWCCD-65-TR-2003/00, Naval Surface Warfare Center Carderock Division, Ship Systems Engineering Station, 5001 S Broad St Philadelphia Pa 19112-5083, February 2003.

[70] Shai, O. and Preiss, K., "Graph theory representations of engineering systems and their embedded knowledge," *Artificial Intelligence in Engineering*, vol. 13, pp. 273–285, 1999.

[71] Simpson, T. W., Mauery, T. M., Korte, J. J., and Mistree, F., "Kriging Models for Global Approximation in Simulation-Based Multidisciplinary Design Optimization," *AIAA Journal*, vol. 39, pp. 2233–2241, December 2001.

[72] Sjöberg, J., *Non-Linear System Identification with Neural Networks*. PhD thesis, Linköping University, S-581 83 Linköping, Sweden, 1995.

[73] Sola, J. and Sevilla, J., "Importance of Input Data Normalization for the Application of Neural Networks to Complex Industrial Problems," in *IEEE Transactions on Nuclear Science*, vol. 44, June 1997.

[74] Spindel, R., S. Laska, J. C.-B., Cooper, D., Hegmann, K., and Hogan, R., "Optimized Surface Ship Manning," Tech. Rep. NRAC-00-1, Naval Research Advisory Committee, 800 North Quincy Street Arlington, VA 22217-5660, April 2000.

[75] Tam, K.-S., "Modeling Approaches for Large-Scale Reconfigurable Engineering Systems," in *Proceedings of World Academy of Science, Engineering and Technology*, vol. 17, pp. 135–140, December 2006.

[76] United States Government Accountability Office, "Challenges Facing the DD(X) Destroyer Program." Report to the Chairman, Subcommittee on Projection Forces, Committee on Armed Forces, House of Representatives, September 2004. GAO-04-973.

[77] van der Merwe, R., Leen, T. K., Lu, Z., Frolov, S., and Baptista, A. M., "Fast Neural Network Surrogates for Very High Dimensional Physics-Based Models in Computational Oceanography," *Neural Networks*, vol. 20, pp. 462–478, 2007.

[78] van Rossaum, G., *Python Tutorial, Release 2.6.5.* Python Software Foundation, March 2010.

[79] Visala, A., "Identification of Wiener-MLP with feedback NOE-model with extended Kalman filter," in *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*, vol. 2, (Anchorage, AK), pp. 1281–1286, May 1998.

[80] Visala, A., Pitkänen, H., and Aarne, H., "Modeling of chromatographic separation process with wiener-mlp representation," *Journal of Process Control*, vol. 11, pp. 443–458, 2001.

[81] Walks, J. P. and Mearman, J. F., "Integrated Engineering Plant," ASNE Reconfiguration and Survivability Symposium, February 2005.

[82] Walters, E. A., Iden, S., Borger, W., and Wampler, B., "INVENT modeling, simulation, analysis and optimization," in *48th AIAA Aerospace Science Meeting Including the New Horizons Forum and Aerospace Exposition*, (Orlando, Florida), 4-7 January 2010.

[83] Wang, J. M., Fleet, D. J., and Hertzmann, A., "Gaussian Process Dynamical Model," in *Proc. Neural Information Processing Systems*, December 2005.

[84] Wellstead, P. E., *Introduction to Physical System Modelling.* Academic Press Ltd., 1979.

[85] Werbos, P., "Backpropagation through time: What it does and how to do it," in *proceedings of the IEEE*, vol. 78, pp. 1550–1560, October 1990.

[86] Westwick, D. and Verhaegen, M., "Identifying MIMO Wiener Systems using Subspace Modeling Identification Methods," *Signal Processing*, vol. 52, pp. 235–258, October 1996.

[87] Williams, R. and Zipser, D., "Experimental analysis of the real-time recurrent learning algorithm," *Connection Science*, vol. 1, no. 1, pp. 87–111, 1989.

[88] Work, R., O'Rourke, R., Labs, E., and McCarthy, J., "Recapitalizing and Modernizing the Navy's Surface Battle-Line." Testimony published by Center for Strategic and Budgetary Assessment, March 2006.

[89] Wu, C.-F. J. and Hamada, M., *Experiments: Planning, Anlysis, and Prameter Design Optimization.* John Wiley & Sons, Inc., 2000.

[90] Zink, M. G., Brown, K., Dalessandro, D., and Longo, D., "Domino - simulating systems of systems." Advanced Automation and Controls R&D, NAVSEA Warfare Centers, Philadelphia, 2009.