INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Design and Evaluation of a Neural Network-Based Controller for an Artificial Heart

by

Martin J. Conlon, B.Eng.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Mechanical and Aerospace Engineering

Department of Mechanical and Aerospace Engineering
Carleton University
Ottawa, Ontario
Canada

© 2000 Martin J. Conlon

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-57723-6

Canada

**The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis**

**DESIGN AND EVALUATION OF A
NEURAL NETWORK-BASED CONTROLLER FOR AN
ARTIFICIAL HEART**

submitted by

Martin J. Conlon, B.Eng.
in partial fulfillment of the requirements for
the degree of Master of Engineering

Thesis Co-supervisor

Thesis Co-supervisor

Chair, Dépt. of Mechanical and Aerospace Engineering

Carleton University
June 12, 2000

Abstract

This thesis presents a neural network-based controller for a ventricular assist device. A thirteenth order mathematical model of the human circulatory system is used for the controller design.

Improvements made to the model include differentiable valve resistances, and a nonlinear venous segment. A backwards Euler-type integration routine was coded in Java to improve simulation performance.

The neural network controller features sixteen nodes and two hidden layers. It has four inputs (current pump output, current systolic fraction, previous diastolic force, and previous systolic force) and two outputs (suggested changes to the systolic and diastolic forces for the next beat).

The differentiable valve resistances provide the same behaviour as the previous threshold function. The nonlinear venous segment is able to model venous collapse at low internal pressures. It also has the positive side effect of improving the model by damping out oscillations in the input cannula.

The controller is able to achieve its design goals, maintaining a peripheral blood flow rate of $4.8 \frac{L}{min}$ unless peripheral resistance drops. (In which case it is trained to increase the blood flow rate).

Acknowledgments

This thesis is the result of a large amount of work, most of it fueled by the bitter alkaloid $C_8H_{10}N_4O_2$. It would not have been completed without the help of many people, and I'd like to express my gratitude to them.

A special thanks must first go to Donald Russell, my thesis co-supervisor, for his guidance and support, and the many hours he spent with me trying to figure out why our "virtual patient" was bleeding to death.

I'd like to thank Tofy Mussivand, my thesis co-supervisor, for his support, and the quick responses to my questions.

Nancy Powell did a good job of shielding me from the administrative headaches associated with completing a Masters degree, and I'd like to thank her for that. Well, that and saving my bacon more than a few times.

I'd like to thank Donald Bailey for de-mystifying Java, and wrapping my head around the idea of object-oriented programming.

Peter Frise encouraged me to consider graduate studies, and I'm extremely grateful for that. This has been a very rewarding experience.

I'd like to thank all of my friends for feigning interest when I prattled on about the simulation, and for tolerating my chronic lateness and absenteeism.

And finally, I'd like to thank my family, for putting up with my odd hours and various idiosyncrasies as well as my sometimes less-than-friendly disposition, while still offering me much-needed encouragement and support.

Contents

Abstract	iii
Acknowledgments	iv
List of Figures	x
List of Symbols	xiii
List of Symbols	xiv
1 Introduction	1
1.1 The Cardiovascular Model	1
1.2 Simulation	1
1.3 Controller Design	2
1.4 Controller Evaluation	2
1.5 Controller Sensitivity	3
2 Literature Review	4
2.1 Neural Networks	4
2.2 Ventricular Assist Devices	4
2.2.1 Control Methods	5
2.3 Cardiovascular Elements	6
2.4 VAD Parameters	7
2.4.1 Diastolic Force	7
2.4.2 Systolic Force Increment	7
2.4.3 Pump Output	8
2.4.4 Beat Rate	8

2.5	Venous Models	9
3	Neural Network Theory	10
3.1	Units	11
3.2	Connections	13
3.3	Organization	13
3.4	Topology	13
3.5	Feed Forward Networks	14
3.6	Training	14
3.6.1	Training Data	16
3.6.2	Preventing Overfitting or Underfitting	17
3.6.3	Training Evaluation	18
3.7	Neurocontrol	18
3.7.1	Teaching Inputs	19
3.7.2	Plant Models	19
3.7.3	Inverse Plant Models	21
3.7.4	Time-delayed Predictions	23
3.7.5	Controller-Critic Models	24
4	The Cardiovascular Model	26
4.1	The Cardiovascular Model	26
4.2	Model Improvements	32
4.2.1	Differentiable resistances	32
4.2.2	Venous Model	33
4.2.3	Integration Scheme	43
4.2.4	Simulation Results	52
5	The Controller	69
5.1	Design	69
5.1.1	VAD Forces	70
5.1.2	Beat Timing	70
5.1.3	Physiological Effects	71
5.2	Implementation	72
5.2.1	Tapped Delay Lines	72
5.2.2	Pump Output Estimation Parameter	73

5.2.3	Systolic Fraction Variation	73
5.2.4	Network Topology	74
5.2.5	Training Strategy	75
5.3	Controller Performance	76
5.3.1	Pump Output	76
5.3.2	Beat Rate	77
5.3.3	Pressure-Volume Loops	77
5.3.4	Changes in Peripheral Resistance	78
5.4	Controller Sensitivity	88
5.4.1	Initial Condition Variations	88
5.4.2	Parameter Variations	103
6	Conclusions and Recommendations	128
6.1	Conclusions	128
6.2	Future Work	129
	References	131
A	Package ai.neural	135
A.1	Classes	136
A.1.1	Class Cluster	136
A.1.2	Class ConnectionGroup	138
B	Package ai.neural.neuron	141
B.1	Classes	142
B.1.1	Class NodeComparator	142
B.1.2	Class Node	143
B.1.3	Class Context	148
B.1.4	Class Neuron	151
B.1.5	Class InputNode	155
B.1.6	Class OutputNode	156
C	Package ai.neural.neuron.functions	158
C.1	Interfaces	159
C.1.1	Interface OutputFunction	159

C.1.2	Interface TransferFunction	159
C.1.3	Interface InputFunction	161
C.2	Classes	163
C.2.1	Class LogSig	163
C.2.2	Class Linear	165
C.2.3	Class TanSig	167
C.2.4	Class Product	169
C.2.5	Class Sum	170
D	Package ai.neural.connection	173
D.1	Classes	174
D.1.1	Class ConnectionComparator	174
D.1.2	Class Connection	175
E	Package ai.neural.network	179
E.1	Classes	180
E.1.1	Class Network	180
E.1.2	Class NetworkComparator	183
E.1.3	Class NetworkTimer	185
F	Package ai.neural.network.functions	186
F.1	Classes	187
F.1.1	Class TrainingFunction	187
F.1.2	Class Backpropagation	188
F.1.3	Class BackpropagationMomentum	190
G	Package numerical.matrix	192
G.1	Classes	193
G.1.1	Class Matrix	193
G.1.2	Class Vector	211
H	Package numerical.workspace	216
H.1	Classes	217
H.1.1	Class Workspace	217

I	Package numerical.integration	220
I.1	Interfaces	221
I.1.1	Interface ODEFunction	221
I.2	Classes	222
I.2.1	Class IntegratorClient	222
I.2.2	Class IntegratorServer	224
I.2.3	Class Integrator	226
J	Package numerical.formatting	233
J.1	Classes	234
J.1.1	Class NumberFormat	234
K	Package numerical.machine	235
K.1	Classes	236
K.1.1	Class Machine	236
L	Package mjc.util	237
L.1	Interfaces	238
L.1.1	Interface BidirectionalEnumeration	238
L.1.2	Interface Comparator	239
L.2	Classes	240
L.2.1	Class Group	240
L.2.2	Class GenericComparator	242
L.2.3	Class Group.Link	243
M	Package graphics.graph	245
M.1	Classes	246
M.1.1	Class GraphFrame	246
M.1.2	Class GraphPane	246

List of Figures

3.1	A simple network with three layers and six units	10
3.2	Artificial neuron	11
3.3	Copying an existing controller	19
3.4	Copying an existing controller using different control inputs	20
3.5	System identification	20
3.6	Control using a system inverse	21
3.7	Identifying a system inverse	22
3.8	Backpropagation of output error	23
3.9	Training an adaptive predictor	24
3.10	Reinforcement learning using a critic	24
4.1	Bond graph model of the human circulatory system	28
4.2	Schematic diagram of the bond graph model.	29
4.3	Schematic representation of the single compliance model	34
4.4	Schematic representation of the dual compliance model	35
4.5	Constitutive relation for the nonlinear compliance element	36
4.6	Bond graph representation of the new venous model	37
4.7	Linear venous volume with respect to time	38
4.8	Nonlinear venous volume with respect to time	39
4.9	Venous resistance with respect to time	40
4.10	Venous pressure difference with respect to time	41
4.11	Inflow and outflow for test of venous collapse	43
4.12	Venous volume with respect to time	44
4.13	Venous resistance with respect to time	45
4.14	Venous pressure difference with respect to time	46
4.15	Arc segment for corner smoothing	52

4.16 Comparison of arterial pressures	53
4.17 Comparison of input cannula pressures	54
4.18 VAD pressure, flow rate and piston force	57
4.19 Pressure gradient and flow rate across the output valve	58
4.20 Output cannula pressure and flow rate	59
4.21 Aorta pressure and flow rate	60
4.22 Arterial pressure and flow rate	61
4.23 Peripheral flow rate	62
4.24 Peripheral venous segment pressure and flow rate	63
4.25 Vena cava pressure and flow rate	64
4.26 Venous pressure drop	65
4.27 Input cannula pressure and flow rate	66
4.28 Pressure gradient and flow rate across input valve	67
5.1 VAD systolic and diastolic force cycle	70
5.2 Effect of P_f and P_{fb} on beat length and systolic fraction	71
5.3 Controller inputs using a tapped delay line	73
5.4 Selected network topology	74
5.5 Pump output under nominal systemic conditions	76
5.6 Pressure-volume loop under nominal conditions	77
5.7 Controller response after an increase in peripheral resistance	78
5.8 Beat length after an increase in peripheral resistance	79
5.9 Pump output after an increase in peripheral resistance	80
5.10 Output cannula pressure after an increase in peripheral resistance	81
5.11 Peripheral flow after an increase in peripheral resistance	82
5.12 Pressure-Volume loop for VAD after an increase in peripheral resistance	83
5.13 Controller response after a decrease in peripheral resistance	84
5.14 Beat length after a decrease in peripheral resistance	85
5.15 Pump output after a decrease in peripheral resistance	86
5.16 Output cannula pressure after a decrease in peripheral resistance	87
5.17 Peripheral flow after a decrease in peripheral resistance	88
5.18 Pressure-Volume loop for VAD after a decrease in peripheral resistance	89
5.19 Pressure in the output cannula after an increase in output cannula blood volume	91
5.20 Peripheral venous volume after an increase in output cannula blood volume	92

5.21 Pump output after an increase in output cannula blood volume	93
5.22 Controller response after an increase in output cannula blood volume	94
5.23 Pressure in the output cannula after a decrease in output cannula blood volume	95
5.24 Peripheral venous volume after a decrease in output cannula blood volume	96
5.25 Controller response after an decrease in output cannula blood volume	97
5.26 Output cannula pressure after an increase in peripheral momentum	98
5.27 Peripheral venous volume after an increase in peripheral momentum	99
5.28 Controller response after an increase in peripheral momentum	100
5.29 Output cannula pressure after a decrease in peripheral momentum	101
5.30 Controller response after a decrease in peripheral momentum	102
5.31 Output valve flow rate for an increase in VAD inertia	104
5.32 VAD Pressure for an increase in VAD inertia	105
5.33 Pressure-volume loop for VAD after an increase in VAD inertance	106
5.34 Pressure-volume loop for VAD after a decrease in VAD inertance	107
5.35 Aorta pressure after and increase in output cannula inertance	108
5.36 Arterial flow rate after an increase output cannula inertance	109
5.37 Pressure-volume loop for VAD after an increase in output cannula inertance	110
5.38 Output valve flow for a large increase in output cannula inertance	111
5.39 Aorta pressure after a decrease in output cannula inertance	112
5.40 Arterial flow rate after a decrease in output cannula inertance	113
5.41 Pressure-volume loop for the VAD after a decrease in output cannula inertance	114
5.42 Aorta pressure after an increase in aorta inertance	115
5.43 Aorta pressure after a decrease in aorta inertance	116
5.44 VAD pressure after an increase in peripheral inertance	117
5.45 Aorta pressure after an increase in peripheral inertance	118
5.46 Arterial flow rate after an increase in peripheral inertance	119
5.47 Output valve flow after an increase in peripheral inertance	120
5.48 VAD pressure after a decrease in peripheral inertance	121
5.49 Output cannula pressure after a decrease in peripheral inertia	122
5.50 Arterial flow rate after a decrease in peripheral inertance	123
5.51 Peripheral venous pressure after an increase in peripheral inertance	124
5.52 Input cannula pressure after an increase in input cannula inertance	125
5.53 Venous flow rates after an increase in input cannula inertance	126

List of Tables

4.1	Model States	31
4.2	Constitutive relations [28], [40]	31
4.3	Nominal resistance and inertance for system elements from [28]	55

List of Symbols

A		Cross-sectional area
A_{k+1}		Number of function computations required
$a_i(t)$		Activation of unit i in step t
α		Learning momentum
C		Linearized element compliance
C_{ao}		Aorta compliance
C_{at}		Arterial compliance
C_{cnin}		Input cannula compliance
C_{cnout}		Output cannula compliance
C_{ht}		VAD compliance
C_{vc}		Vena cava compliance
C_{vn}		Peripheral venous compliance
Δ_{pout}		Normalized difference in pump output
Δ_{sys}		Normalized variation in systolic fraction
ΔP_{in}		Pressure difference across inlet valve
ΔP_{ot}		Pressure difference across outlet valve
ΔP_{vs}		Pressure difference across venous segment
Δw_{ij}		Change in weight of link from unit i to j
d		Offset distance
δ_j		Error for unit j
δx		x-offset
δy		y-offset
ϵ		Error tolerance
ϵ_{k+1}		Error estimate for step $k + 1$
η		Learning rate

F_{in}	Input force
$\mathbf{f}$	State equations
f'_j	Derivative of neuron transfer function
$g(\dots)$	An arbitrary function (see text)
H_k	Next step size
h	Step size
$h(\dots)$	An arbitrary function (see text)
I	Linearized element inertance
I_{ao}	Aorta inertance
I_{cnin}	Input cannula inertance
I_{cnout}	Output cannula inertance
I_{ht}	Piston inertance
I_{ic}	Input cannula inertance
I_{oc}	Output cannula inertance
I_{pe}	Peripheral inertance
L	Length
MSE	Mean squared error
m	Slope
net_j	The net input into unit j
$o_i(t)$	Output of unit i in step t
o_{ij}	Actual output of unit j for pattern i
$offset$	Transition pressure offset
ω	Residual beat frequency
P	Internal pressure
$P(Q)$	Pressure drop
P_{ao}	Aorta pressure
P_{ar}	Arterial pressure
P_{ht}	VAD internal pressure
P_{ic}	Input cannula internal pressure
P_{oc}	Output cannula internal pressure
P_{out}	Pump output
P_{res}	Residual beat pressure
P_v	Peripheral (linear) venous pressure

P_{vc}	Vena cava pressure
P_{vn}	Peripheral (linear) venous pressure
$\Phi_c(P)$	Element compliance
$\Phi_i(Q)$	Element inertance
$\Phi_r(Q)$	Flow resistance
p	Linearized momentum
$p(Q)$	Momentum
Q	Flow rate
Q_{vc}	Vena cava flow rate
Q_{vn}	Peripheral venous flow rate
R	Linearized flow resistance
$R(x)$	Flow resistance
R_{ao}	Aorta resistance
R_{cnin}	Input cannula resistance
R_{cnot}	Output cannula resistance
R_{ht}	Piston resistance
R_{ic}	Input cannula resistance
R_{in}	Inlet valve resistance
R_{max}	Maximum resistance
R_{min}	Minimum resistance
R_{oc}	Output cannula resistance
R_{ot}	Outlet valve resistance
R_{pe}	Peripheral resistance
R_{vn}	Venous resistance
R_{vs}	Venous resistance
r	Corner radius
ρ	Fluid density
S_i	Normalized input
S_{f1}	Flow source
S_{f2}	Flow source
SSE	Sum of squared errors
$slope$	Transition slope
σ	Standard deviation

t	Simulation time
t_{beat}	Beat length
t_{ij}	Desired output of unit j for pattern i
t_j	Desired output of unit j
θ_j	Threshold (bias) of unit j
$\mathbf{U}$	A series of control inputs
$u(t)$	Control input at time t
V	Linearized internal volume
$V(P)$	Internal volume
V_{max}	Maximum VAD volume
V_{min}	Minimum VAD volume
V_v	Peripheral venous volume
V_{vc}	Vena cava volume
V_{vad}	VAD volume
W_k	Work for step k
w_{ij}	Weight of the link from unit i to unit j
X_i	Input data
$\bar{X}$	Mean of input data
$\mathbf{x}(t)$	System state at time t
$\mathbf{y}$	State vector

Chapter 1

Introduction

The goal of this thesis is to provide a neural network controller for a ventricular assist device. Controller design is performed using a numerical cardiovascular model. Along with the design of the controller itself, improvements were made to the model and the integration scheme.

1.1 The Cardiovascular Model

The human cardiovascular system was modeled using the bond graph energy method, producing the system illustrated in Figure 4.1, which results in thirteen first order differential equations. These thirteen equations, which are described in Chapter 4, can be integrated numerically with respect to time to predict the behaviour of the cardiovascular system.

The bond graph model consists of an output cannula, aorta, lumped arterial system, peripheral arterioles, peripheral veins, lumped vena cavae, input cannula, and artificial heart. The artificial heart is modeled as in [40], as a cylindrical chamber with a piston and a compliance.

Improvements to the previous model [28] include a nonlinear vena cava element, and a pressure-dependent venous flow resistance, which is able to model venous collapse at low internal pressures.

1.2 Simulation

The state equations were implemented in Java, and integrated using a custom integration routine, described in Section 4.2.3. The integration routine is a variant of the modified mid-

point method, altered to be more tolerant of stiff systems. The supporting matrix and vector routines were also coded in Java, along with the neural network code used by the controller. All simulation code, the matrix library, the integration routine, and the neural network library are documented in the Appendix.

1.3 Controller Design

A neural network-based controller was designed to improve upon the standard full-fill, full-eject method of control, adding more flexibility to its response to systemic changes. The controller alters the systolic and diastolic forces to maintain proper pump output. “Proper” pump output is taken to be $4.8 \frac{L}{min}$ [10], unless the peripheral resistance drops, in which case the controller is instructed to increase pump output.

Pump output is calculated on a beat-by-beat basis, based on the beat duration and the fixed artificial heart volume, which is taken to be $80ml$. The controller inputs include the current pump output, the current systolic fraction, the previous diastolic force, and the previous systolic force. The outputs are the suggested changes to the diastolic and systolic forces for the next beat.

The controller is trained online, while the circulatory parameters are varied. Training is stopped after the controller has restored proper pump output.

1.4 Controller Evaluation

The controller performance was evaluated under nominal conditions, and under varying peripheral conditions. The controller provides the desired pump output even when the peripheral resistance is doubled, and behaves as a full-fill, full-eject controller when the peripheral resistance is halved, increasing the pump output to $5.6 \frac{L}{min}$. Under nominal conditions the controller maintains a beat frequency of $1Hz$, providing a pump output of $4.8 \frac{L}{min}$.

Results of the controller evaluation are discussed in Section 5.3.4.

1.5 Controller Sensitivity

The controller sensitivity was tested by varying various systemic parameters, namely: the heart inertance; output cannula inertance; aorta inertance; and input cannula inertance, as well as the initial conditions.

The controller sensitivity study, and parameter variations are discussed in detail in Section 5.4.

Chapter 2

Literature Review

2.1 Neural Networks

Neural networks are adaptive structures consisting of large numbers of simple processing nodes connected via directional links. The units operate in parallel, and the network function is determined by connection strength, topology, and nodal transfer functions. Training algorithms are used to adjust connection weights based on data, so that neural networks “learn” by example.

Neural networks are used for classification and function approximation problems which require tolerance to noise or imprecision, and have been applied to the control of arc furnaces [38], aircraft [18], and robots [8],[26]. They have been effective in controlling a variety of nonlinear dynamic systems [7],[23],[24],[35].

2.2 Ventricular Assist Devices

There are two main types of “artificial hearts”: total artificial hearts (TAHs) and ventricular assist devices (VADs) [11],[16].

Total artificial hearts are designed to pump the blood in the pulmonary loop, as well as the blood entering the circulatory system, and have two artificial ventricles. The input cannula to the TAH’s right ventricle is attached to the right atrium of the natural heart, or the vena cava. The TAH’s left ventricle is attached to the natural heart’s left ventricle and the aorta via cannulae.

Such a complex mechanism is often unnecessary, as many cases of myocardial dysfunction involve a damaged left ventricle, with a reasonably healthy right ventricle. These cases can be treated by a ventricular assist device, which is designed to provide assistance to the natural heart.

The VAD input is attached to either the left atrium or the left ventricle, and its output cannula is attached to the aorta. The VAD fills under pressure from the left atrium, energizes the blood by some means, and ejects it into the aorta, assisting the left ventricle in providing the high pressure blood output into the circulatory system.

2.2.1 Control Methods

Fixed Rate Control

The fixed rate method of control is more suited to in-hospital use, where the ventricular assist device's beat rate and systolic fraction can be controlled under the supervision of a doctor. The fixed rate controller cycles the VAD a preset number of times per minute, producing a constant pump output. This type of controller is unable to respond to changes in circulatory system parameters.

Full-Fill, Full-Eject Control

A full-fill, full-eject controller employs positional sensors (often Hall-effect sensors) to trigger its systolic and diastolic cycles. The minimum and maximum VAD volumes are selected to prevent clotting while still maintaining proper pump output, and the controller is able to adjust slightly to changes in systemic parameters as the preload pressure dictates the fill rate. The systolic and diastolic pressures in the artificial heart are fixed with this type of controller [13],[30].

Fuzzy Logic Control

Fuzzy logic is a rule-based or "expert" method of control. Proper control actions are programmed into the controller, which makes its decisions using a variant on conventional logic (if-then) that allows for input noise and gradual transitions between sets. (Classical sets have crisp boundaries, and classical "if" statements return "true" or "false"; fuzzy "if" statements do not.)

Fuzzy logic has successfully been applied to the control of artificial hearts [21],[28].

Other Control Methods

Both traditional and digital control methods have been applied to ventricular assist devices [9],[22].

2.3 Cardiovascular Elements

The human cardiovascular system can be viewed as a very complicated hydraulic network that can be broken down into a variety of subsystems, such as the pulmonary system, the arterial system and the venous system. The behaviour of each of these can be modeled using a few basic elements and concepts.

An element's compliance (which is the inverse of stiffness) defines the change in internal pressure for a given change in internal volume, as in (2.1), which linearizes to (2.2).

$$V(P) = \Phi_c(P) \quad (2.1)$$

$$V = C \cdot P \quad (2.2)$$

An element's inertance gives the inertia of the fluid in the segment, or relationship between the fluid's momentum and flow rate, as in (2.3), which linearizes to (2.4). In hydraulic elements it is a function of cross sectional area, length and fluid density, as in (2.5).

$$p(Q) = \Phi_i(Q) \quad (2.3)$$

$$p = I \cdot Q \quad (2.4)$$

$$I = \frac{\rho L}{A} \quad (2.5)$$

Flow resistance through an element can be defined as the relationship between flow rate and pressure drop, as in (2.6), which can be linearized to (2.7).

$$P(Q) = \Phi_r(Q) \quad (2.6)$$

$$P = R \cdot Q \quad (2.7)$$

2.4 VAD Parameters

2.4.1 Diastolic Force

The force applied to the piston during diastole is responsible (along with the preload pressure in the input cannula) for the fill rate in the VAD. If the diastolic force is too high, the VAD fills slowly. If it is lower, the VAD fills more quickly. A controller can maintain a reasonable beat rate by manipulating the diastolic force.

The preload pressure (and the diastolic force) also influence the VAD volume at the end of systole; if the preload pressure is low and the VAD internal pressure or the diastolic force is high, the input valve may close prematurely, as the VAD uses unidirectional passive flow valves. In a fixed rate control situation, this premature closing is not a major concern, but if the controller relies on VAD filling (as a full-fill, full-eject controller does), a premature valve closing, leading to an under-filled VAD is a major concern. The systolic ejection phase may not be triggered.

Note that the preload pressure itself cannot easily be changed; only the diastolic force can be altered.

2.4.2 Systolic Force Increment

The systolic force increment is the difference between the diastolic force and the systolic force. During systole, or the ejection phase, an incremental force is added to the diastolic force in order to increase the internal pressure in the VAD and trigger the valves. As the internal VAD pressure rises, the input valve closes (due to the high negative pressure gradient) and the output valve opens, allowing blood to be ejected from the heart. (The valves do not necessarily trigger simultaneously; their operation is passive, and dependent on both internal and external VAD pressures.)

The systolic force increment governs the ejection phase of the pump cycle. It dictates both the ejection pressure of the blood and the length of the systolic fraction. If the systolic force increment is large, the systolic fraction is short, as blood is ejected quickly, and the ejection pressure is high. A high ejection pressure might be necessary to maintain pump output when faced with high flow resistance in the periphery, but persistent hypertension (high blood pressure)

can damage the aorta and arterial system. Ejection pressures are best kept below 175mmHg , and preferably in the $120 - 130\text{mmHg}$ range [10].

The systolic force increment has very little effect on the VAD preload pressure, because the arterial and venous segments are effectively decoupled. The peripheral blood vessels have extremely small diameters, which translates to a high inertia.

2.4.3 Pump Output

Pump output is defined as in (2.8), and varies depending on the beat length and the ejected volume.

$$P_{out} = \frac{V_{vad}}{t_{beat}} \quad (2.8)$$

An increase in beat length (or a decrease in the number of beats per minute) lowers the pump output. A decrease in ejected volume also lowers the pump output.

Human cardiac requirements vary depending on age and size, but typically fall within the $4.5 - 5.3 \frac{L}{min}$ range [1]. The required pump output also varies with activity level, age and size [5].

2.4.4 Beat Rate

As mentioned above, the beat rate, or number of beats per minute, directly affects pump output. Methods of varying beat rate depend on the VAD control strategy employed. A fixed rate controller presents the simplest challenge: beat rate can be altered directly by the controller. Varying beat rate with a full-fill, full-eject controller is more difficult. The beat rate of such a controller is dependent on the preload pressure in the input cannula, the diastolic pressure in the VAD, the systolic pressure in the VAD and the afterload pressure in the output cannula.

Of these four parameters, only the systolic and diastolic pressures are controllable, and a full-fill, full-eject control system must vary the beat rate indirectly, by varying the diastolic force and systolic force increment, in the fashion mentioned earlier.

2.5 Venous Models

A preliminary treatment of the venous system can employ a linear compliance model, but the larger veins behave as thin-walled, flexible tubes, and can collapse under low internal pressures. This venous collapse results in an increased flow resistance, dramatically altering systemic conditions.

A thin-walled, flexible tube can be modeled using two nonlinear compliance elements connected by a single constrictive segment [20]. The constrictive, or resistance element is dependent on the transmural pressure across the upstream segment, because when the tube is fully open the upstream and downstream pressures are similar. When the tube collapses, the upstream pressure increases.

The behaviour of the peripheral veins is linear, and a model of the complete venous system can employ a linear compliance in series with one or more nonlinear elements [5],[27].

Chapter 3

Neural Network Theory

This chapter is included to provide a brief overview of neural network theory as background for the reader.

Artificial neural networks consist of large numbers of simple processing units, which work in parallel, and are connected via directed links, which carry numeric data between units, as shown in Figure 3.1.

Figure 3.1: A simple network with three layers and six units

In this way, knowledge is distributed among the many processors making these connectionist models very damage, fault, and noise tolerant.

3.1 Units

Neurons in artificial neural networks are divided into three classes. The units whose activations form the problem input to the network are called input units. The neurons whose outputs represent the network output are called output units. The remaining neurons are referred to as hidden units because they are not “visible” from outside the network. Hidden units are responsible for the majority of the processing performed by an artificial neural network.

All neurons, no matter what their classification, accept inputs and produce outputs. The actual information processing performed by a neural network is dependent on the transfer functions of the individual neurons.

Figure 3.2: Artificial neuron

Each neuron’s transfer function can be broken down into two components: an activation function, used to compute the activation level, or degree of excitation, of the unit, and; an output function, used to compute the neuron’s output for a given activation level.

Activation functions are usually chosen to produce bounded output, taking into account the net input into the unit during the current time step and the threshold or bias of the unit. Many of the most popular activation functions have a sigmoidal shape. An example of this is the logistic function:

$$a_j(t) = \frac{1}{1 + e^{-(\sum_i w_{ij} o_i(t) + \theta_j)}} \quad (3.1)$$

where:

- $a_j(t)$ activation of unit j in step t
- $o_i(t)$ output of unit i in step t
- w_{ij} weight of the link from unit i to unit j
- θ_j threshold (bias) of unit j
- i index for some unit in the network
- j index for another unit in the network ($j > i$)

Another popular activation function (which does *not* produce bounded output) is the identity function:

$$a_j(t) = \left(\sum_i w_{ij} o_i \right) + \theta_j \quad (3.2)$$

The unifying feature for the many activation functions is that their derivatives are easily found, allowing efficient training of the network.

Like the unit activation functions, the output functions are quite consistent across networks. The two most common are the identity (3.3) and clipping (3.4) functions:

$$o_i(t) = a_i(t) \quad (3.3)$$

or

$$o_i(t) = \begin{cases} -1 & \text{if } a_i(t) < -1 \\ a_i(t) & \text{if } -1 \leq a_i(t) \leq 1 \\ 1 & \text{if } a_i(t) > 1 \end{cases} \quad (3.4)$$

where:

- $a_i(t)$ activation of unit i in step t
- $o_i(t)$ output of unit i in step t

The identity output function is normally used with all activation functions known to produce bounded output (such as the logistic function). The clipping function is used with activation functions not producing bounded output (such as the identity activation function).

3.2 Connections

Connections between units allow the transfer of activation. The unit at which the connection originates is called the source unit while the other is called the target unit. The source and target unit can be the same, allowing recursive connections, but there can be only one connection between a given source unit and a given target. (Redundant connections are not permitted).

Each connection has an assigned weight which determines the effect the source has on the target. If the weight is negative, the source inhibits the target; if it is positive, the source excites the target.

Links are not bidirectional; information is only permitted to flow from the source to the target.

3.3 Organization

Neurons are usually grouped into layers according to their classification (as input, hidden or output.) While there can only be one input layer and one output layer, there can be multiple hidden layers in a network. Networks with one hidden layer are typically only able to solve linear classification problems, while those with multiple layers can solve non-linear problems [36].

Within the individual layers there is no organization according to connection type. That is, inhibitory and excitatory links can co-exist between two layers.

Connection weights, which determine whether a link is inhibitory or excitatory, are initialized to random values and then modified by training. The initial random values have a noticeable effect on the final state of any given link.

3.4 Topology

The numbers of input and output units are normally fixed, as they are dependent on the problem being solved, whether that be pattern recognition or system control. Selecting the number of hidden layers, is also a fairly straightforward task; non-linear problems require at

least two hidden layers, and keeping the number of layers to a minimum reduces processing and training time.

Choosing an appropriate number of hidden units, on the other hand, is an arduous task. The number of hidden units required depends on:

- The number of input and output units.
- The number of sets of training data.
- The amount of noise in the targets.
- The complexity of the problem to be learned.
- The hidden unit activation function.
- The training algorithm.

In practice, the only way to determine the ideal number of hidden units is by experimentation. If there are too few units, the network will not learn, and the validation error will be large. If there are too many hidden units, the network will overlearn the training data, and will again be unable to generalize.

3.5 Feed Forward Networks

Feed forward networks are perhaps the most common type of neural network, and are so-named because all of the activation signals feed forward through the network; there are no recurrent connections. This class of network can be trained using either gradient descent algorithms, or derivative-free methods [15],[36].

3.6 Training

Most network learning algorithms are based on the *Hebbian learning rule* [15],[36], which updates links between neurons proportional to the rate at which they fire together:

$$\Delta w_{ij} = \eta \cdot g(a_j(t), t_j) \cdot h(o_i(t), w_{ij}) \quad (3.5)$$

where:

w_{ij}	weight of link from unit i to unit j
η	learning rate
$a_j(t)$	activation of unit j in step t
t_j	the desired output of unit j
$o_i(t)$	output of unit i at time t
$g(\dots)$	arbitrary function, dependent on the activation and the desired output
$h(\dots)$	arbitrary function, dependent on the output of unit i and the current weight of the link

The problem with the generalized Hebbian rule is that the weights grow in an unconstrained fashion. For this reason, a modified version of the rule which incorporates weight normalization, using a *regularization function* is often used.

If the network error is plotted in weight space, it appears as a series of peaks and valleys. Only one of the many weight combinations represents the global minimum for the network error, but there are normally several local error minima. The goal of all training algorithms is to minimize the square of the network error by finding the appropriate combination of weights. (The square of the network error is chosen because the value is always positive).

The simplest learning rule which applies the Hebb principle is the backpropagation method of learning (3.6).

$$\Delta w_{ij} = \eta \cdot \delta_j \cdot o_i \quad (3.6)$$

$$\delta_j = \begin{cases} f'_j(net_j)(t_j - o_j) & \text{if unit } j \text{ is an output unit} \\ f'_j(net_j) \sum_k \delta_j w_{jk} & \text{if unit } j \text{ is a hidden unit} \end{cases} \quad (3.7)$$

where:

δ_j	error for unit j
f'_j	derivative of neuron transfer function
net_j	net input into unit j
i, j, k	index of units ($i < j < k$)

To speed the rate of learning, a momentum term can be added to the standard backpropagation equation, yielding (3.8). The purpose of this momentum term is to make the current

weight change a function of the previous weight change, which not only speeds learning but also prevents the weight oscillations which are common when there is a very narrow minimum. (Meaning the “valley” surrounding the minimum of the network error space is very narrow). The momentum term can also help escape a local minimum.

$$\Delta w_{ij}(t + 1) = \eta \cdot \delta_j \cdot o_i + \alpha \cdot \Delta w_{ij}(t) \quad (3.8)$$

The backpropagation algorithm and the various other gradient descent algorithms require that the neuron transfer function be differentiable (as mentioned earlier.) There are also several other training methods which are not derivative-based, such as simulated annealing, genetic algorithms and Monte Carlo methods.

3.6.1 Training Data

During training, a network is normally supplied a “learning” input and a “teaching” output. The training algorithm is used to adjust the network weights to minimize some “objective” function, which is normally representative of the network error. This may be a sum of the total error function and the regularization function. If no weight normalization is used (meaning the regularization function is zero), the objective function is equal to either the total or average error function.

Sets of training data must be carefully chosen to provide good coverage of the population, while avoiding too much repetition. Typically three sets of data are employed: a training set of examples used for learning; a validation set of examples used to tune network parameters (such as the number of hidden layer neurons); and a test set of examples used to assess the performance of the network.

Normally network input and output data is standardized, by subtracting some measure of position and dividing by some measure of scale. This prevents one input (or output) from dominating training by virtue of its magnitude alone, and speeds network learning. Several learning algorithms, most notably gradient descent algorithms based on the Hebbian learning rule, are very sensitive to input scaling, because an ill-conditioned Hessian matrix slows learning considerably. (The Hessian matrix is the second derivative of the network weights).

One of the most useful ways to standardize inputs is to make the mean zero and the standard deviation one as in (3.9).

$$S_i = \frac{X_i - \bar{X}}{\sigma} \quad (3.9)$$

Output scaling is also important if one output's range is large compared to another output's, especially if the error function is scale-sensitive (like the mean squared error function.) The target variables should be scaled so that their variability reflects their importance, rather than their physics. Typically target outputs are standardized to the same range, or the same standard deviation if they are of equal importance.

If suitable training data is not available beforehand (as is typically the case in control applications), data can be accumulated and shifted and scaled appropriately as training progresses. This *online* training normally requires a flexible method of training evaluation, or error calculation.

If the neural network is being trained to approximate a smooth function, adding statistical noise to the training data may improve generalization, resulting in a network that is less susceptible to errors in the input data. In order for this to be effective, the noise should have zero mean and finite variance, and care must be taken, because noise in the target values increases the chance of overfitting the data [2].

3.6.2 Preventing Overfitting or Underfitting

Neural networks, like any other nonlinear estimator, can suffer from either overfitting or underfitting. In other words, the network either doesn't generalize well, as is the case with overfitting, or doesn't produce coherent results, a condition which is termed underfitting.

The best way to prevent overfitting is to have a large amount of training data, something which is often impractical. (In order to prevent overfitting, there must be approximately thirty times more training sets than there are weights in the network, or, for noise free data, at least five times as many training sets.)

An alternative to using a large amount of training data is to train using the "early stopping" technique, as follows:

- Divide the data into training and validation sets.
- Use a large number of hidden units.
- Use very small initial random weights.
- Use a slow learning rate.
- Stop training as soon as the validation error starts to increase.

This training technique can be revised and extended somewhat. If one is careful, it is not necessary to use a “large” number of hidden neurons, and in a control system application, where online learning can be used, the effective size of the training set can be quite large, as older training data is discarded on a regular basis.

3.6.3 Training Evaluation

The network performance can be evaluated by examining the sum of squared error, as in (3.10), and the mean squared error of the outputs (3.11).

$$SSE = \sum_{i=1}^n \sum_{j=1}^m (t_{ij} - o_{ij})^2 \quad (3.10)$$

$$MSE = \frac{SSE}{n} \quad (3.11)$$

where:

n number of patterns

m number of output units

3.7 Neurocontrol

Neurocontrol systems go beyond the traditional pattern recognition or system identification applications of neural networks, as the control networks are able to influence their input signals. This influence brings with it new challenges in training, because training data is not necessarily available ahead of time. Typically some variant of online training is used to train a neural network control system.

3.7.1 Teaching Inputs

If a suitable controller is already available, a neural network can be trained to approximate this controller, as illustrated in Figure 3.3. The benefits of this sort of neurocontroller might

Figure 3.3: Copying an existing controller

not be readily apparent; if there is already a suitable controller, why change things? First, the existing controller may be expensive or impractical to use. Second, the neural network controller doesn't necessarily have to be trained using the same inputs as the existing controller. If the existing controller relies on control inputs that are difficult to obtain, the neurocontroller can be trained to produce the same control behaviour using system indicators that are easier to obtain, as illustrated in Figure 3.4.

3.7.2 Plant Models

By training a network using learning inputs and teaching outputs as illustrated in Figure 3.5, one can develop a neural network model of the given plant. While this might seem somewhat redundant at first glance, the value of this neural plant model will become clear in the next section.

Figure 3.4: Copying an existing controller using different control inputs

Figure 3.5: System identification

3.7.3 Inverse Plant Models

If the inverse transfer function for a plant can be determined, the output from this model can be used as the control input for the actual system, and the controller input then becomes the desired plant output, as illustrated in Figure 3.6.

Figure 3.6: Control using a system inverse

If the order of the plant is known and all of the states can be measured, we have:

$$\mathbf{x}(t+1) = \mathbf{f}(\mathbf{x}(t), u(t)) \quad (3.12)$$

where:

$\mathbf{x}(t+1)$ state at time $t+1$

$\mathbf{x}(t)$ state at time t

$u(t)$ control input at time t

Or, in general:

$$\mathbf{x}(t+n) = \mathbf{F}(\mathbf{x}(t), \mathbf{U}) \quad (3.13)$$

where:

n order of the plant

In other words, given a series of control inputs, $\mathbf{U}$, the plant will move from $\mathbf{x}(t)$ to $\mathbf{x}(t + n)$ in n time steps. Assuming the inverse plant model exists, $\mathbf{U}$ can be expressed as a function of $\mathbf{x}(t)$ and $\mathbf{x}(t + n)$:

$$\mathbf{U} = \mathbf{G}(\mathbf{x}(t), \mathbf{x}(t + n)) \quad (3.14)$$

Since (3.14) does not always have an analytical solution, it is often easier to train a neural network to approximate the inverse plant dynamics.

By supplying the network with learning inputs and teaching outputs as illustrated in Figure 3.7, the network is trained to model the inverse behaviour of the system. Unfortunately, many

Figure 3.7: Identifying a system inverse

plant models are not injective mappings, which means that many different inputs can produce the same outputs. The network then attempts to map a singular input to many different target responses, and this leads to an averaging of sorts, and not necessarily an inverse plant model [17]. As well, determining an inverse plant model, while fairly straight forward with a linear system, is quite a bit more complex when the target system is nonlinear. Nevertheless, this technique has been applied to the control of nonlinear systems [34],[37].

If a (regular) plant model is available, it can be used in conjunction with the actual plant and an inverse system model to train the controller by backpropagating the output error through *both* neural network models in turn, as illustrated in Figure 3.8.

Figure 3.8: Backpropagation of output error

If the network succeeds in modeling the inverse plant dynamics, then given a current system state, $\mathbf{x}(t)$, and a desired future state, $\mathbf{x}_d(t+n)$, the network can generate a series of control actions $\hat{\mathbf{U}}$ designed to bring the system to the desired state after n time steps.

$$\hat{\mathbf{U}} = \hat{\mathbf{G}}(\mathbf{x}(t), \mathbf{x}_d(t+n)) \quad (3.15)$$

Minimization of the network error $\|\mathbf{U} - \hat{\mathbf{U}}\|^2$, does not necessarily imply minimization of the system error $\|\mathbf{x}_d(t) - \mathbf{x}(t)\|^2$.

3.7.4 Time-delayed Predictions

A neural network can be supplied with time delayed inputs and current outputs, as illustrated in Figure 3.9. In this way, it is trained to predict current outputs given previous inputs, but once trained can be used to predict future outputs given current inputs, by removing the delay blocks.

Figure 3.9: Training an adaptive predictor

3.7.5 Controller-Critic Models

Neural network controllers lend themselves well to reinforcement learning, whereby a critic evaluates the controller's performance and the controller adjusts accordingly, as illustrated in Figure 3.10.

Figure 3.10: Reinforcement learning using a critic

This “critic” (which can be a human, or another control system, for example) critiques the controller’s effectiveness and returns an evaluation signal to the controller. The training algorithm can adjust the controller based on the evaluation signal.

Chapter 4

The Cardiovascular Model

As mentioned previously, the human circulatory system is very complex. Good references on the topic include [4], [12], [39] and [40].

4.1 The Cardiovascular Model

The cardiovascular system was modeled using the bond graph energy method[19], producing the system illustrated in Figure 4.1. This bond graph yields thirteen first order nonlinear differential equations, (4.1)-(4.13), which can be integrated numerically with respect to time, to predict the behaviour of the cardiovascular system.

These thirteen state equations, along with their supporting constitutive relations, (4.14)-(4.25), were implemented in Java.

A (perhaps) more familiar schematic representation using hydraulic elements is shown in Figure 4.2. This figure makes the compliance elements' configuration slightly more obvious, which will become important when the simulation performance is discussed.

As can be seen in Figure 4.2, the simplified cardiovascular system model (further detailed in [28] and [40]) can be grouped into seven segments, namely the VAD, the output cannula, the aorta, the arterial system, the peripheral venous system, the vena cavae, and the input cannula.

The VAD consists of a piston, a compliance element, and two passive valves, which are modeled as nonlinear resistances. The output cannula is modeled using a linear compliance element, a linear resistance element and an inertance element. Similarly, the aorta is modeled using a nonlinear compliance element, a linear resistance element and an inertance element. The arterial system is modeled using a single compliance element. The venous system will be discussed in more detail, but consists of an inertance element, a linear resistance element, a linear compliance element (comprising the “peripheral venous system”), a nonlinear resistance element and a nonlinear compliance element (which form the “vena cava system”). The input cannula is very similar to the output cannula (although the parameters are slightly different). It consists of an inertance element, a linear resistance element and a linear compliance.

The initial conditions for the simulation, outlined in Table 4.1, are based on those found in [1],[5], and [28]. The simulation was run to steady state, using a fixed beat rate, allowing the startup transients to die out.

Figure 4.1: Bond graph model of the human circulatory system

Figure 4.2: Schematic diagram of the bond graph model.

$$\dot{p}_3 = F_{in} - \frac{R_{ht}}{I_{ht}} \cdot \dot{x}_1 - A \cdot P_{ht}(V_6) \quad (4.1)$$

$$\dot{V}_6 = \frac{P_{ic}(V_{36}) - P_{ht}(V_6)}{R_{in}(\Delta P_{in})} + \frac{p_3 \cdot A}{I_{ht}} - \frac{P_{ht}(V_6) - P_{oc}(V_{10})}{R_{ot}(\Delta P_{ot})} \quad (4.2)$$

$$\dot{x}_1 = \frac{p_3}{I_{ht}} \quad (4.3)$$

$$\dot{V}_{10} = \frac{P_{ht}(V_6) - P_{oc}(V_{10})}{R_{ot}(\Delta P_{ot})} - \frac{p_{13}}{I_{oc}} \quad (4.4)$$

$$\dot{p}_{13} = P_{oc}(V_{10}) - \frac{R_{oc}}{I_{oc}} \cdot p_{13} - P_{ao}(V_{15}) \quad (4.5)$$

$$\dot{V}_{15} = \frac{p_{13}}{I_{oc}} - \frac{p_{18}}{I_{ao}} \quad (4.6)$$

$$\dot{p}_{18} = P_{ao}(V_{15}) - \frac{R_{ao}}{I_{ao}} \cdot p_{18} - P_{ar}(V_{21}) + P_{res} \cdot \cos(\omega \cdot t) \quad (4.7)$$

$$\dot{V}_{21} = \frac{p_{18}}{I_{ao}} - \frac{p_{24}}{I_{pe}} \quad (4.8)$$

$$\dot{p}_{24} = P_{ao}(V_{21}) - \frac{R_{pe}}{I_{pe}} \cdot p_{24} - P_v(V_{27}) - P_{res} \cdot \cos(\omega \cdot t) \quad (4.9)$$

$$\dot{V}_{27} = \frac{p_{24}}{I_{pe}} - \frac{\Delta P_{vs}}{R_{vs}(\Delta P_{vs})} \quad (4.10)$$

$$\dot{V}_{31} = \frac{P_v(V_{27}) - P_{vc}(V_{31})}{R_{vs}(\Delta P_{vs})} \quad (4.11)$$

$$\dot{V}_{36} = \frac{p_{33}}{I_{ic}} - \frac{P_{ic}(V_{36}) - P_{ht}(V_6)}{R_{in}(\Delta P_{in})} \quad (4.12)$$

$$\dot{p}_{33} = P_{vc}(V_{31}) - \frac{R_{ic}}{I_{ic}} \cdot p_{33} - P_{ic}(V_{36}) \quad (4.13)$$

$$P_{ht}(V_6) = \frac{V_6}{28} - 0.178 \quad (4.14)$$

$$P_{ic}(V_{36}) = \frac{V_{36}}{32} - 2.62 \quad (4.15)$$

$$P_{oc}(V_{10}) = \frac{V_{10}}{32} - 5.24 \quad (4.16)$$

$$P_{ao}(V_{15}) = \frac{10}{9.81 \cdot 0.0135} \cdot (0.0105 \cdot V_{15}^3 - 0.2543 \cdot V_{15}^2 + 3.1069 \cdot V_{15} + 1.4208) \quad (4.17)$$

$$P_v(V_{27}) = \frac{V_{27}}{400} - 4.62 \quad (4.18)$$

State		Description	Units	Nominal
$\dot{p}_3$	$\dot{y}_1$	Pump momentum	$N \cdot s$	0.0
$\dot{V}_6$	$\dot{y}_2$	VAD chamber volume	cm^3	6.4
$\dot{x}_1$	$\dot{y}_3$	Piston position	cm	0.0
$\dot{V}_{10}$	$\dot{y}_4$	Output cannula volume	cm^3	187.0
$\dot{p}_{13}$	$\dot{y}_5$	Output cannula momentum	$\frac{N \cdot s}{cm^2}$	0.0
$\dot{V}_{15}$	$\dot{y}_6$	Aorta volume	cm^3	19.9
$\dot{p}_{18}$	$\dot{y}_7$	Aorta momentum	$\frac{N \cdot s}{cm^2}$	0.0
$\dot{V}_{21}$	$\dot{y}_8$	Arterial volume	cm^3	429.8
$\dot{p}_{24}$	$\dot{y}_9$	Peripheral momentum	$\frac{N \cdot s}{cm^2}$	0.89
$\dot{V}_{27}$	$\dot{y}_{10}$	Peripheral venous volume	cm^3	2275.1
$\dot{V}_{31}$	$\dot{y}_{11}$	Vena cavae volume	cm^3	800
$\dot{V}_{36}$	$\dot{y}_{12}$	Input cannula volume	cm^3	118.2
$\dot{p}_{33}$	$\dot{y}_{13}$	Input cannula momentum	$\frac{N \cdot s}{cm^2}$	0.0

Table 4.1: Model States

Function	Description	Input	Output Units
P_{ht}	VAD pressure	V_6	$\frac{N}{cm^2}$
P_{ic}	Input cannula pressure	V_{36}	$\frac{N}{cm^2}$
P_{oc}	Output cannula pressure	V_{10}	$\frac{N}{cm^2}$
P_{ao}	Aorta pressure	V_{15}	$\frac{N}{cm^2}$
P_v	Peripheral venous pressure	V_{27}	$\frac{N}{cm^2}$
P_{vc}	Vena cavae pressure	V_{31}	$\frac{N}{cm^2}$
ΔP_{in}	Pressure difference across inlet valve	V_6, V_{36}	$\frac{N}{cm^2}$
ΔP_{ot}	Pressure difference across outlet valve	V_6, V_{10}	$\frac{N}{cm^2}$
ΔP_{vs}	Pressure difference in venous segments	V_{27}, V_{31}	$\frac{N}{cm^2}$
R_{in}	Inlet valve resistance	V_6, V_{36}	$\frac{N \cdot s}{cm^5}$
R_{ot}	Outlet valve resistance	V_6, V_{10}	$\frac{N \cdot s}{cm^5}$
R_{vs}	Venous segment resistance	V_{27}, V_{31}	$\frac{N \cdot s}{cm^5}$

Table 4.2: Constitutive relations [28], [40]

$$P_{vc}(V_{31}) = \begin{cases} -0.98 - (200 - V_{31}) \cdot 500 & V_{31} < 200 \\ \frac{-0.98 + (V_{31} - 200)}{500} & 200 \leq V_{31} \leq 800 \\ -0.22 + (V_{31} - 800) \cdot 500 & 800 < V_{31} \end{cases} \quad (4.19)$$

$$\Delta P_{in} = P_{ic} - P_{ht} \quad (4.20)$$

$$\Delta P_{ot} = P_{ht} - P_{oc} \quad (4.21)$$

$$\Delta P_{vs} = P_v - P_{vc} \quad (4.22)$$

$$R_{in}(\Delta P_{in}) = 0.0008 + \frac{800 - 0.0008}{1 + e^{-20 \cdot \Delta P_{in}}} \quad (4.23)$$

$$R_{ot}(\Delta P_{ot}) = 0.0008 + \frac{800 - 0.0008}{1 + e^{-20 \cdot \Delta P_{ot}}} \quad (4.24)$$

$$R_{vs}(\Delta P_{vs}) = 0.002 + \frac{0.04 - 0.002}{1 + e^{-4 \cdot \Delta P_{vs} - 3.8}} \quad (4.25)$$

4.2 Model Improvements

Three main improvements were made to the previous [28] model and simulation:

- The constitutive relations for the valves were changed to be differentiable (and smooth.)
- The venous system was altered to more closely resemble the human circulatory system, allowing for venous collapse at low internal pressures.
- The simulation was re-coded in Java and a new backwards Euler-type integration scheme [33] was implemented to improve simulation efficiency.

4.2.1 Differentiable resistances

Three elements in the system model, namely the input and output valves, and the venous “resistance”, are modeled as nonlinear resistances. From an implementation point of view, the most efficient way to model this sort of behaviour is through the use of a hard limit, or threshold function (4.26).

$$R(x) = \begin{cases} R_{min} & x < threshold \\ R_{max} & x > threshold \end{cases} \quad (4.26)$$

Unfortunately, this resistance model dramatically affects integration time, because of the severe parameter change at the threshold. A smoother function, which gradually transitions from the low to high resistance (or vice versa) is much more desirable from a numerical integration standpoint. The sigmoidal function (4.27) displays the same characteristics as the threshold function (it has two asymptotic limits), but it has a controllable transition threshold, and is differentiable. The function is much more computationally intensive than the hard limit function, but it is hoped that the savings in integration time (due to a reduction in the number of steps required) will offset the cost of the computation.

$$R(x) = R_{min} + \frac{R_{max} - R_{min}}{1 + e^{-2 \cdot slope \cdot (x - offset)}} \quad (4.27)$$

where:

slope controls the transition slope

offset the bias offset

4.2.2 Venous Model

The larger veins in the body, such as the various vena cavae, can be modeled as thin-walled, flexible tubes. As such, their pressure-volume constitutive relations are nonlinear, but for a first approximation, the nonlinear effects can be neglected. Obviously for more accurate results, these must be considered.

The most important nonlinear effect, from a modeling point of view, is the tendency towards collapse at low internal pressures. This collapse dramatically increases the resistance to blood flow. Since the venous system normally operates at very low internal pressures, it is important to consider the possibility of venous collapse, especially when modeling an implanted device, which is able to apply suction to the venous system.

The Venous System

The venous system is responsible for the return of blood to the heart, and consists of a series of blood vessels with increasingly larger diameters, namely the venules, the veins, and the vena cavae.

The venules are very similar to capillaries, in that they are microscopic vessels, but unlike the capillaries, which branch off from the arteries, the venules join together to return deoxygenated blood from the cells to the veins.

The veins are larger in diameter than the venules, and are composed of three layers: an inner layer of epithelial cells, which are in contact with the blood; a middle layer of elastic fibers and smooth muscle; and an outer layer of fibrous tissue. Since the veins operate at very low internal pressure, the veins in many parts of the body have valves to prevent backflow, or flow away from the heart. Although their internal pressure is quite low, the mean cross sectional area of a vein is quite small (0.2cm^2) [41], and collapse of the smaller veins is not a large concern. As well, the total cross sectional area for the veins is quite large (80cm^2), and nonlinearities in the pressure-volume relationship are likely negligible [40].

The vena cavae are larger veins, which return blood to the heart, or, in this case, the input cannula for the ventricular assist device, as the remnant heart and the pulmonary loop are not modeled. These large vessels have the same structure as the smaller veins, consisting of three distinct layers of tissue. Their average cross sectional area is large, at 7cm^2 , and nonlinear behaviour at both high and low internal pressures must be taken into consideration.

Single Compliance Model

For a first approximation, the entire venous system can be modeled as in Figure 4.3, using a single compliance element whose constitutive relation is the volume response of the venous system to changes in internal pressure. This is how the venous system was treated prior to this work [40],[28].

Figure 4.3: Schematic representation of the single compliance model

The single compliance element was assumed to have a constant value of $3500 \frac{\text{cm}^5}{\text{N}}$, with an offset of -0.4 , yielding (4.28) a pressure of $0.25 \frac{\text{N}}{\text{cm}^2}$ for the nominal blood volume of 2250ml . This value is slightly higher than what would be found in the literature, [1],[5], but is necessary

for reasons that will become clear in the “Simulation Results” section (section 4.2.4). Briefly, in this circulatory system model, the venous pressure is solely responsible for filling the VAD. The VAD requires a pressure of approximately 15mmHg in order to fill (which would be delivered by the remnant heart), while the typical venous pressure is only 5mmHg . The venous pressures have been adjusted accordingly.

$$P_v(V) = \frac{V}{3500} - 0.4 \quad (4.28)$$

Dual Compliance Model

The dual compliance model is based on the realization that nonlinear effects will be negligible in many of the smaller vessels in the venous system. Only the larger vena cavae, which contain a relatively small percentage of the the total venous blood volume, will exhibit noticeable nonlinearities.

The venous system was modeled using two compliance elements, as illustrated in Figure 4.4. The first, linear, compliance models the behaviour of the smaller “peripheral” veins, which contain (nominally) $2L$ of the system’s blood volume. The second, nonlinear, compliance models the lumped behaviour of the larger veins, which contain a total of $0.8L$ of the blood volume.

Figure 4.4: Schematic representation of the dual compliance model

Splitting the venous system up into two separate compliance elements has an added advantage: by inserting a nonlinear resistance element, we can model the collapse of the larger veins at low internal pressures.

The constitutive relations for all three elements are inter-related. The linear compliance element is similar to that found in the single compliance model. The pressure at nominal volume was chosen as $0.38 \frac{N}{cm^2}$, and the slope was adjusted to give a pressure of $-0.87 \frac{N}{cm^2}$ at a volume of $1.5L$. These values are somewhat arbitrary, but based on those found in the literature [5],[10]. This leads to equation (4.29):

$$\begin{aligned} P_v(V) &= \Phi_c^{-1}(V) \\ &= \frac{V}{400} - 4.62 \end{aligned} \quad (4.29)$$

To help with model refinements, a piecewise function, shown in Figure 4.5, was used to model the nonlinear compliance. The “corner pressure” at nominal volume was set to $0.22 \frac{N}{cm^2}$, and the internal pressure before collapse was set to $-0.98 \frac{N}{cm^2}$. These values are (again) slightly higher than those commonly found in literature.

Figure 4.5: Constitutive relation for the nonlinear compliance element

Figure 4.6: Bond graph representation of the new venous model

$$P_{vc}(V) = \Phi_C^{-1}(V)$$

$$= \begin{cases} -0.98 - (200 - V_{31}) \cdot 500 & V_{31} < 200 \\ -0.98 + \frac{V_{31} - 200}{500} & 200 \leq V_{31} \leq 800 \\ 0.22 + (V_{31} - 800) \cdot 500 & 800 < V_{31} \end{cases} \quad (4.30)$$

The resistance function was chosen to allow a flowrate of $50 \frac{ml}{s}$ when the vessel was collapsed and $400 \frac{ml}{s}$ when it was fully open. The offset, or transition pressure difference, was chosen as $3.8 \frac{N}{cm^2}$, based on the two compliance functions.

$$\Delta P_{vs} = P_v - P_{vc} \quad (4.31)$$

$$R_{vs}(\Delta P_{vs}) = 0.002 + \frac{0.04 - 0.002}{1 + e^{-4 \cdot (\Delta P_{vs} - 3.8)}} \quad (4.32)$$

The bond graph shown in Figure 4.6 yields two state equations, (4.33) and (4.34). These can be integrated numerically while altering the inflow (and outflow) to (and from) the system, producing conditions similar to those which might occur in the human body.

$$\dot{V}_v = S_{f1} - \frac{\Delta P}{R_{vs}} \quad (4.33)$$

$$\dot{V}_{vc} = \frac{\Delta P}{R_{vs}} - S_{f2} \quad (4.34)$$

Steady State Behaviour

To obtain the steady-state behaviour of the improved venous model, the two states were integrated for a period of 10s, with sinusoidal input and output flow rates (4.35), representative

of the blood flow typical in the body.

$$S_{f1} = S_{f2} = 50 \cdot \sin(2 \cdot \pi \cdot t) + 50 \quad (4.35)$$

The resultant volume in the linear segment (with respect to time) is shown in Figure 4.7. The volume in the nonlinear segment (again, with respect to time) is plotted in Figure 4.8 and the venous resistance and pressure difference across the two venous segments are plotted in Figures 4.9 and 4.10 (respectively).

Figure 4.9 is included to show that the venous flow resistance does not change over the course of the simulation, in spite of the pressure “spikes” visible in Figure 4.10.

These spikes are due to integration error. The constitutive relation for the non-linear venous system has two very abrupt transitions, and a small change in vena cava volume can result in a large change in pressure. As well, the second derivative is discontinuous, which has a negative impact on the variable step-sizing algorithm.

Figure 4.7: Linear venous volume with respect to time

Figure 4.8: Nonlinear venous volume with respect to time

Figure 4.9: Venous resistance with respect to time

Figure 4.10: Venous pressure difference with respect to time

A comparison of the two venous models will be deferred until the “Simulation Results” section (4.2.4) because the single compliance model does not yield a state equation that is independent of net inflow.

Venous Collapse

As mentioned previously, an important characteristic of thin-walled tubes is their tendency to collapse at low internal pressures, resulting in an increased flow resistance. To test this behaviour, the inflow and outflow for the venous model were varied as shown in Figure 4.11.

From $t = 0s$ to $t = 30s$ the inflow was held at $100 \frac{ml}{s}$ while the outflow was held at $150 \frac{ml}{s}$, resulting in a net outflow of $50 \frac{ml}{s}$. At $t = 30s$ the situation was reversed, giving a net inflow of $50 \frac{ml}{s}$. At $t = 60s$ the inflow was reduced to $100 \frac{ml}{s}$, giving no net inflow or outflow. These conditions were maintained until $t = 70s$, at which point the inflow and outflow were raised to $200 \frac{ml}{s}$. At $t = 80s$ both flow rates were reduced to $150 \frac{ml}{s}$, which was held until the end of the simulation.

The behaviour of the system is as expected. Initially both segments empty together, as seen in Figure 4.12, but the nonlinear segment (containing a smaller portion of the total blood volume) reaches a critically low pressure at approximately $t = 21s$. At this point, the internal pressure in the nonlinear segment drops dramatically, and that portion of the veins collapses. This collapse can be seen in Figure 4.13, as the flow resistance increases at that time.

At $t = 30s$, the net positive flowrate begins to fill the system. This is initially apparent in Figure 4.12, as the linear venous portion begins to fill, and in Figure 4.13, as the flow resistance drops, but not readily visible in the nonlinear segment (Figure 4.12), as the internal pressure in the nonlinear portion is still near the critical threshold value, and a small increase in internal volume yields a relatively large increase in internal pressure. In other words, the nonlinear venous segment is “inflating” for the initial 10s of filling.

Both segments continue filling under the net positive inflow until $t = 60s$, at which time the net inflow is zero, as the inflow and outflow are held at the same value. At $t = 70s$ the net inflow is held at zero, but the inflow and outflow rates are doubled from $100 \frac{ml}{s}$ to $200 \frac{ml}{s}$, which is larger than the typical volume flow rates found in the human body [41]. This results in a

shift in the systemic volume as the pressures across the two segments adjust. Note that there is no change in venous resistance (Figure 4.13.)

The flowrates are decreased to $150 \frac{ml}{s}$ at $t = 80s$, which results in another volume shift, but again no change in venous flow resistance.

The venous pressure difference, plotted in Figure 4.14 displays the same integration artifacts as Figure 4.10.

Figure 4.11: Inflow and outflow for test of venous collapse

4.2.3 Integration Scheme

One of the great difficulties with numerical simulation is obtaining accurate numbers within a reasonable timeframe. Integration time can be reduced by lowering the error tolerance, but the results can be less accurate.

Figure 4.12: Venous volume with respect to time

Figure 4.13: Venous resistance with respect to time

Figure 4.14: Venous pressure difference with respect to time

The previous [28] circulatory system model was twelfth order (with one dependent state) and highly nonlinear, and the addition of the improved venous model adds another state and yet another nonlinearity. With simulation runs for the previous system taking upwards of 4 hours on a 175MHz R10000 to generate 60 seconds worth of data, it became clear that the system was testing the limits of the Runge-Kutta routine.

Another factor affecting performance was the interpreted (vs. compiled) Matlab code in which the simulation was written. Using Matlab's ODE45 integration routine, which is a hybrid Runge-Kutta method, with 4th and 5th order function approximations and adaptive step sizing, eliminated the need to write a custom integration routine and supporting matrix and vector code. Unfortunately, the interpreted code has a negative impact on integration performance.

Java

In order to improve performance without sacrificing the portability enjoyed with the Matlab code, it was decided to re-write the simulation in Java, and code a custom integration routine which was more tolerant of stiff systems. Java was chosen as the development language because the code will compile unchanged on any system, and the compiled code itself will run on any system. Even languages such as C and FORTRAN do not provide this degree of portability; the numerical data types in these languages vary across platforms, and certain types of behaviour (notably, overflow, underflow, and division by zero) do not have a well defined treatment.

As well, the Java libraries provide arbitrary precision arithmetic in both integers and decimals, which is true for very few other languages. They also provide platform independent graphics capabilities, excellent networking (for distributed processing), exception handling, and are multi-threaded.

The price for this portability is a decrease in execution speed, but benchmark tests using C++ and Java showed a minimal performance advantage for the native C++ code, ranging between 5 and 10 percent. (Initially the simulation was developed in two parallel versions: C++ and Java, until the performance of the Java version was assured. The two languages are quite similar.)

The Semi-implicit Extrapolation Method

There are numerous integration schemes which treat stiff equations effectively. These include Rosenbrock methods, Predictor-Corrector methods and Gear routines [33],[43]. The integration routine selected was the semi-implicit discretization method developed by Bader and Deuffhard [33], which is based on the Bulirsch-Stoer method and applies the modified midpoint rule. This integration routine is very efficient when working with stiff equations, and lends itself well to distributed processing, something which can be taken advantage of as the simulation becomes more complex and computationally intensive.

The starting point is an implicit form of the midpoint rule:

$$\mathbf{y}_{n+1} - \mathbf{y}_{n-1} = 2 \cdot h \cdot \mathbf{f} \left(\frac{\mathbf{y}_{n+1} + \mathbf{y}_{n-1}}{2} \right) \quad (4.36)$$

where:

$\mathbf{y}$ vector of states

$\mathbf{f}$ the state equations (function)

h the step size

To convert this equation to semi-implicit form, we must linearize the right hand side about $\mathbf{f}(\mathbf{y}_n)$:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot \mathbf{f}(\mathbf{y}_n) \cdot \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right]^{-1} \quad (4.37)$$

Multiplying by unity gives:

$$\mathbf{y}_{n+1} = \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right]^{-1} \cdot \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right] \cdot \left(\mathbf{y}_n + h \cdot \mathbf{f}(\mathbf{y}_n) \cdot \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right]^{-1} \right)$$

Substituting (4.36) and expanding yields:

$$\begin{aligned} \mathbf{y}_{n+1} &= \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right]^{-1} \cdot \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right] \cdot \mathbf{y}_{n-1} + \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right]^{-1} \cdot 2 \cdot h \cdot \mathbf{f}(\mathbf{y}_n) \\ &\quad - \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right]^{-1} \cdot 2 \cdot h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \cdot \mathbf{y}_n \end{aligned}$$

Finally, collecting terms gives:

$$\left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right] \cdot \mathbf{y}_{n+1} = \left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right] \cdot \mathbf{y}_{n-1} + 2 \cdot h \cdot \left[\mathbf{f}(\mathbf{y}_n) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \cdot \mathbf{y}_n \right] \quad (4.38)$$

The semi-implicit midpoint rule (4.38) is used in combination with a special first step, the semi-implicit Euler step (4.36) and a special smoothing last step:

$$\bar{\mathbf{y}}_n \equiv \frac{1}{2} (\mathbf{y}_{n+1} + \mathbf{y}_{n-1}) \quad (4.39)$$

On its own this midpoint rule is not terribly useful. Its power comes when it is called repeatedly, using increasingly finer substeps and the resultant values of the states extrapolated to an infinitely fine substep. In other words, a series of finite substeps are used to move from t to $t + H$, where H is quite a large number, using a finite number of substeps to compute the state values at $t + H$. These values are extrapolated in a tableau to arrive at an answer corresponding to an infinite number of substeps.

Extrapolation using tableau expansion is often called *Neville's algorithm*. Polynomial or rational functions of increasing degree are fitted to the data, and an error estimate is obtained using the differences between consecutive approximations. Neville's algorithm provides a recurrent way to fit higher order polynomials to the data. The expansion is carried out in a tableau.

The step size sequence, by Deuffhard [33], used in the semi-implicit extrapolation method is:

$$n = 2, 6, 10, 14, 22, 34, 50, 70, 98, \dots \quad (4.40)$$

Each number differs from its predecessor by the smallest multiple of four that makes the ratio of successive terms $\leq \frac{5}{7}$.

A rational function extrapolation (4.41) was employed to predict the resultant states after an infinite number of substeps. Rational functions can be superior to polynomials because of their ability to model functions with poles [33].

$$R_{i(i+1)\dots(i+m)} = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 \cdot x + \dots + p_\mu \cdot x^\mu}{q_0 + q_1 \cdot x + \dots + q_\nu \cdot x^\nu} \quad (4.41)$$

Each expansion of the rational function tableau is treated as an estimate of the error and is used to control the step sizing as per Deuffhard [33]. The estimate of a new step size, H_k to

obtain convergence in a fixed tableau column, k is:

$$H_k = H \left(\frac{\epsilon}{\epsilon_{k+1,k}} \right)^{\frac{1}{2k+1}} \quad (4.42)$$

where:

- ϵ_{k+1} error estimate for step $k + 1$
- ϵ required error tolerance

For the purposes off efficiency we compute a “work coefficient” for a step, which assumes the bulk of the computational time is spent evaluating the state equations and their Jacobian. For n_k subdivisions, the number of function evaluations, A , can be found from:

$$\begin{aligned} A_1 &= n_1 + 1 + N \\ A_{k+1} &= A_k + n_{k+1} \end{aligned} \quad (4.43)$$

where:

- n number of subdivisions for the integration step
- A_k number of function evaluations required to compute tableau column k
- N number of states

The work per step becomes:

$$W_k = A_{k+1} \cdot \left(\frac{\epsilon_{k+1,k}}{\epsilon} \right)^{\frac{1}{2k+1}} \quad (4.44)$$

The work coefficients can be calculated throughout the course of the integration, and the step size can be adjusted accordingly. A factor of safety, 4, was used to scale the error estimate.

Matrix Conditioning

During the course of the integration, the inverse of (4.45) must be calculated repeatedly, and when integrating a stiff system, this matrix may become singular.

$$\left[\mathbf{1} - h \cdot \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \right] \quad (4.45)$$

In order to make the integration scheme more robust, singular value decomposition (SVD) was used to compute the matrix inverse. This inversion scheme, while more computationally intensive than LU-decomposition (for example), is able to successfully compute the inverse of singular (or near-singular) matrices. As a side effect, the matrix's singular values can be used

to determine its condition number. As the condition number (the ratio between the largest and smallest singular values) becomes large, the matrix is declared ill-conditioned, and verging on singularity.

By monitoring the matrix condition number throughout the course of the integration, one can vary the step size to help maintain stability. In the coded integration routine, the step size was reduced by a factor of the current stepping sequence number (4.40) if the reciprocal of the matrix condition number bordered on the machine epsilon. The machine epsilon is representative of the relative error of a floating-point approximation. Intuitively, it is the smallest number which, added to 1, yields an answer different from 1 [43].

Jacobian Calculations

The semi-implicit extrapolation routine requires that the Jacobian of the state equations be calculable, and this poses a slight problem, because the derivative of the piecewise venous constitutive relation is discontinuous. Two solutions to this problem were implemented:

- The corners of the function were rounded by computing an arc segment tangent to the linear slopes.
- A numerical Jacobian routine was devised.

If the slopes of consecutive segments are reciprocals of each other, the arc calculation becomes a trivial matter, as the arc center lies on the bisector of the corner angle, which is in turn directly related to the slope.

From Figure 4.15:

$$\theta = \frac{\pi}{2} - 2 \cdot \left(\tan^{-1} \left(\frac{1}{m} \right) \right) \quad (4.46)$$

$$d = r \cdot \left(\cos \left(\frac{\theta}{2} \right) + \tan \left(\frac{\theta}{2} \right) \cdot \sin \left(\frac{\theta}{2} \right) \right) \quad (4.47)$$

$$\Delta y = d \cdot \cos \left(\frac{\pi}{4} \right) \quad (4.48)$$

$$\Delta x = d \cdot \sin \left(\frac{\pi}{4} \right) \quad (4.49)$$

Figure 4.15: Arc segment for corner smoothing

To compute a numerical Jacobian, one can employ the finite difference method, incrementing the individual states by some small number, computing the state derivatives using this new value, and dividing the result by this same small number.

The choice of the “small number” is dependent on the system being analyzed. Some smooth functions do not have rapidly changing derivatives, while other functions have discontinuous slopes. In the coded integration routine, it was decided to base the increment on a scalar multiple of the current step size, yielding the following:

$$\left. \begin{aligned} \mathbf{y}_{tmp} &= \mathbf{y} \\ \Delta y &= \epsilon \cdot h \cdot y_{tmp}(i) \\ y_{tmp}(i) &= y_{tmp}(i) + \Delta y \\ \frac{\partial \mathbf{y}}{\partial t_i} &= \mathbf{f}(\mathbf{y}_{tmp}) \end{aligned} \right\} i = 1, 2, 3, \dots, n$$

The numerical Jacobian was used for the simulation, because the arc segment introduced small discontinuities at the corners, owing to round-off errors.

4.2.4 Simulation Results

The Venous Models

When the simulation is run under nominal conditions, the venous system pressure remains relatively high, and the vena cavae in the new venous model do not collapse. This precludes

any noticeable pressure difference between the two venous systems. The effect on the arterial system is also minimal, as can be seen in Figure 4.16, but there is a noticeable effect on the input cannula pressures, plotted in Figure 4.17.

Figure 4.16: Comparison of arterial pressures

The previous system model suffered from a very noticeable water hammer-like effect, due to the interaction between the cannula compliance and the fluid inertia. The large pressure oscillations during systole, immediately after the input valve closes, are readily visible in Figure 4.17. The input cannula pressure is very oscillatory because the effective flow resistance of either cannula is zero. As the water hammer-like effect is not present in the physical system, increasing the flow resistance of the input cannula element was considered. This would have damped the pressure oscillations, but would not have been a justifiable or defensible change to the system model. (The flow resistance of the cannulae is indeed *very* close to zero, if not zero.)

The output cannula did not display the same pressure oscillations as the input cannula.

Figure 4.17: Comparison of input cannula pressures

Updating the venous model has a positive effect on the input cannula behaviour, and the water hammer disappears almost entirely. One artifact can still be seen in the input cannula pressure plot, as a little spike, the peak of which coincides exactly with the first pressure oscillation of the old model.

Nominal Circulatory Simulation

The resistance and inertance values for the various system elements, most of which remain unchanged from [28], are given in Table 4.3, and the initial conditions for the thirteen system states were included earlier, in Table 4.1.

Variable	Description	Nominal Value	Units
I_{ht}	VAD inertance	0.0038	$\frac{N \cdot s^2}{cm^5}$
R_{ht}	VAD resistance	0.47	$\frac{N \cdot s}{cm^5}$
I_{oc}	Output cannula inertance	0.0000263	$\frac{N \cdot s^2}{cm^5}$
R_{oc}	Output cannula resistance	0.0	$\frac{N \cdot s}{cm^5}$
R_{ot}	Output valve resistance	0.0008-800	$\frac{N \cdot s}{cm^5}$
I_{ao}	Aorta inertance	0.00003401	$\frac{N \cdot s^2}{cm^5}$
R_{ao}	Aorta resistance	0.0011	$\frac{N \cdot s}{cm^5}$
I_{pe}	Peripheral inertance	0.0275	$\frac{N \cdot s^2}{cm^5}$
R_{pe}	Peripheral resistance	0.0125	$\frac{N \cdot s}{cm^5}$
I_{ic}	Input cannula inertance	0.0000132	$\frac{N \cdot s^2}{cm^5}$
R_{ic}	Input cannula resistance	0.0	$\frac{N \cdot s}{cm^5}$
R_{in}	Input valve resistance	0.0008-800	$\frac{N \cdot s}{cm^5}$

Table 4.3: Nominal resistance and inertance for system elements from [28]

To aid presentation the simulation was run for 2.0s, under nominal conditions, using a fixed beat rate. The systolic fraction was set to 0.3s, and the beat length was set to 1.0s. This makes the transition between systole and diastole easily recognizable, while a full-fill, full-eject control strategy might have affected the beat length and the systolic fraction. *Note that the controller presented later in this work is a variant on the full-fill, full-eject strategy and thus was not used for this two second simulation.*

After the thirteen state equations, (4.1)-(4.13), are integrated with respect to time, the data requires some post-processing to compute quantities of interest, such as internal pressures and flow rates.

The constitutive relations, (4.14)-(4.25) can be used to calculate the internal pressures of the various system elements, and the pressure differences across the two valves and venous resistance. The flow rates through the valves can be easily computed, given the pressure difference and the valve resistance at time t .

The pressure and flow rate for the VAD itself are graphed along with piston force in Figure 4.18. The piston force is set to $40N$ during diastole and $0N$ during systole, with the beat frequency outlined above. The VAD pressure reaches a maximum of $135.3mmHg$ during systole and holds relatively constant at $10mmHg$ during diastole. It should be noted that (4.2) is not representative of the volume of blood in the VAD, but rather the volume of blood in the VAD compliance element itself, and thus the flow rate into this element is not the flow rate into the circulatory system. This distinction will become more obvious when we examine the arterial segment.

As can be seen in Figure 4.19, the pressure gradient across the output valve is positive during the systolic phase (allowing flow through the valve) and negative during the diastolic phase (restricting flow through the valve.) The pressure differences and flow rates are in good agreement with those outlined in [28], indicating that the valve's new, smooth constitutive relations closely match the previous threshold function.

A close inspection of the output valve flow rate (Figure 4.19) reveals a very small ($5 \times 10^{-5} \frac{L}{min}$) backflow across the valve during diastole. This backflow, which is not visible on the graph because of its small magnitude, is expected as the resistance of the closed valve, while large, is not infinite.

The output cannula pressure and flow rate are plotted in Figure 4.20, and closely match the aorta pressure and flow rates, plotted in Figure 4.21. This behaviour is due to the low flow resistance and relatively short length of the output cannula. The pressure in the output cannula peaks at $122.7mmHg$ during systole and drops to $80.6mmHg$ during diastole. The pressure in the aorta peaks at $124.6mmHg$ during systole, slightly higher than the pressure in the output

Figure 4.18: VAD pressure, flow rate and piston force

Figure 4.19: Pressure gradient and flow rate across the output valve

cannula, because of the pressure oscillations in the aorta. Due to its higher compliance (i.e., lower stiffness), the output cannula has less pronounced pressure waves.

Figure 4.20: Output cannula pressure and flow rate

The “dicrotic notch” is visible just after the switch to diastole, at $t = 0.39s$, in the pressure plots of both the output cannula and the aorta, although the pressure oscillations in the aorta tend to mask the behaviour somewhat.

Both the output cannula and the aorta have similar flow rates, peaking at $17 \frac{L}{min}$ during systole and dropping to $2 \frac{L}{min}$ during diastole, for an average flow rate of $4.3 \frac{L}{min}$ for the duration of the beat.

While the pressures in the output cannula and the aorta rise and fall in unison, there is a slight lag between the aorta and the arterial pressures. This is due to the combined length of the output cannula, which is modeled by the inertia element I_{oc} , and the aorta, which is modeled by the inertia element I_{ao} .

Figure 4.21: Aorta pressure and flow rate

The arterial pressure and flow rate, shown in Figure 4.22, are much smoother than those of the preceding elements, but upon close inspection, evidence of the dicrotic notch still remains. The arterial pressure peaks at 113.1mmHg during systole and drops to 78.5mmHg during diastole, with an average flow of $-1.3\frac{\text{L}}{\text{min}}$. This represents the flow *into* the arterial compliance element, rather than the blood flow through the arteries themselves.

Figure 4.22: Arterial pressure and flow rate

The peripheral system element, representing the organs, small capillaries and venules, has a very high inertia, and the flow rate increases for the first few beats, as can be seen in Figure 4.23, but settles after 10s of simulation to a nominal flow rate of approximately $4.5\frac{\text{L}}{\text{min}}$.

Maintaining a constant peripheral blood flow rate is one of the controller design goals (to be discussed in Chapter 5) because the organs and cells rely on a steady supply of oxygenated blood to function. If the blood flow is restricted for any length of time, cells begin to die, and reduced flow over a prolonged period can lead to organ failure.

Figure 4.23: Peripheral flow rate

The pressure and flow rate in the peripheral (linear) venous segment are plotted in Figure 4.24, and the pressure and flow rate in the non-linear venous segment are plotted in Figure 4.25. The linear venous segment has a slightly higher pressure than the non-linear segment, as would be expected. The pressure drop across the venous resistance, plotted in Figure 4.26, is minimal because the vena cavae are fully open and the volume flow rate is relatively low. The fully open (or “inflated”) non-linear segment can be seen in the plot of the non-linear venous pressure. The pressure drops slightly at $t = 0.75s$, or approximately half a second after the end of systole.

Figure 4.24: Peripheral venous segment pressure and flow rate

As outlined earlier, the pressures in the two venous segments are higher than would normally be expected because the remnant heart is not modeled, and the venous system empties directly into the VAD. In the proper configuration the VAD would be filled by the right atrium of the heart, and would have an inflow pressure of $10 - 20 \text{ mmHg}$. The venous system would operate at its regular pressure of $0 - 10 \text{ mmHg}$.

Figure 4.25: Vena cava pressure and flow rate

Figure 4.26: Venous pressure drop

Rather than model the heart and the pulmonary loop, it was decided to include the right atrium of the heart in the venous model as a lumped parameter.

The internal pressure and flow rate in the input cannula are plotted in Figure 4.27. The pressure is seen to peak at 33.2mmHg , and drops to a minimum pressure of 16.0mmHg .

Figure 4.27: Input cannula pressure and flow rate

The input cannula flow rate exhibits some sharp oscillations, due to the water hammer effect as the input valve closes, but the pressure oscillations are not severe, owing to the high compliance of the input cannula.

The pressure difference and flow rate across the input valve are plotted in Figure 4.28. During the systolic phase there is a large negative pressure gradient across the input valve, and the flow resistance is at its maximum. During diastole the valve opens under a positive pressure gradient, and the flow rate is seen to peak at $8.8 \frac{\text{L}}{\text{min}}$ before settling to a nearly constant value of $6.4 \frac{\text{L}}{\text{min}}$ for the duration of the diastolic period. Although this flow rate might seem high, one

must keep in mind that the VAD fills for only 70% of the simulation time, and the input valve flow rate corresponds exactly to the systemic flow rate of $4.5 \frac{L}{min}$ seen in the peripheral system elements.

Figure 4.28: Pressure gradient and flow rate across input valve

Integration Routine Performance

For purposes of comparison, the simulation was run using the previous [28] state equations with both the Matlab ODE45 integration routine and the Semi-implicit Extrapolation routine coded in Java. It was decided to use the previous state equations, rather than the state equations outlined in this work because the (extremely stiff) non-linear venous model by itself proved a substantial challenge for the ODE45 routine, requiring large amounts of processing time, and hundreds of thousands of integration steps to produce 20s of simulation data, while the Java routine could produce the same results in a fraction of a second using only a few thousand integration steps.

Using the previous state equations and an error tolerance of 1×10^{-6} , the ODE45 requires 1312s of processing time to produce 5s of simulation data, while the Java routine requires only 10.4s of processing time to produce the same 5s of simulation results. It should again be noted that the Java routine not only runs substantially faster (in terms of integration steps taken per second), but it also takes fewer steps throughout the course of the integration, bettering the ODE45 routine by a factor of 100. (To produce two seconds of simulation data, the ODE45 routine required 112,000 integration steps, while the Java routine took 1000 steps).

Chapter 5

The Controller

The controller goal can be summarized simply: provide a fresh supply of oxygenated blood to the body's organs and tissues. Implicit in this statement are several caveats, which are important to recognize:

- The cardiovascular output should be maintained at approximately $4.8 \frac{L}{min}$, as required by the body.
- The pressures in the aortic and arterial segments must not exceed normal levels, approximately $120mmHg$.
- The number of beats per minute should be maintained in a reasonable range, typically 40-80 beats per minute.
- The VAD should react to physiological changes in the circulatory system, and respond accordingly.

5.1 Design

There are four parameters which can be adjusted to provide control of the pump output and VAD performance. These are the systolic force, the diastolic force, the beat length, and the systolic fraction. The goal of this work is to improve on the full-fill, full-eject method of control, outlined in Section 2.2.1.

5.1.1 VAD Forces

The base, or diastolic force, is applied to the piston during the fill stage, and a pumping, or systolic increment is added to this force during the ejection stage of a given beat. This scenario is outlined in Figure 5.1, with the base force designated P_{fb} and the force increment designated P_f .

Figure 5.1: VAD systolic and diastolic force cycle

The nominal diastolic force was set at $5N$ (or $19.5mmHg$, given a nominal piston area of $19.27cm^2$), and the nominal force increment was set to $30N$ (or $116.8mmHg$.)

5.1.2 Beat Timing

The beat length in a regular full-fill, full-eject controller is determined by the fill rate and ejection rate, which in turn depend on physiological parameters, as well as the systolic and diastolic forces. One can change the beat length or the systolic fraction by modifying the two control forces, within reason. The systolic force should not be made to exceed $170mmHg$ on a regular basis, or damage to the circulatory system may result.

A full-fill, full-eject controller has a fixed minimum and maximum heart volume with which it operates. These volumes are normally tuned to prevent clotting between beats while still maintaining a large ejected fraction of blood volume with each beat. This control strategy

requires positional information on the VAD piston, if only at the extreme ends of its travel (i.e., top dead and bottom dead center.)

While the switch from systole to diastole (and vice versa) is predetermined from a volume point of view, it is not fixed in time. If the VAD fills more slowly due to lower venous pressure, the beat length is automatically extended, or if the aorta internal pressure is lower the systolic fraction may be reduced as the blood volume is ejected more quickly. In short, a supervisory controller could modify the beat length and systolic fraction by varying the base and incremental forces appropriately. This is illustrated in Figure 5.2.

Figure 5.2: Effect of P_f and P_{fb} on beat length and systolic fraction

5.1.3 Physiological Effects

Although it should be readily apparent from this simplified lumped parameter model, the circulatory system is exceedingly complex, with many factors affecting blood flow. Choosing proper controller responses becomes difficult as many similar parameter variations require different VAD responses. For example, a nominal blood flow rate of $4.8 \frac{L}{min}$ is desired, even when flow resistance in the periphery is increased. In other words, the VAD should increase the pumping force slightly if the peripheral resistance rises. Conversely, a decrease in peripheral resistance (as happens when a person exercises or gets cold) indicates a need for a higher blood flow, rather than a simple maintenance of flow rate.

The natural heart has chemical signals to help it distinguish between the various physical states, but the VAD controller has no such indicator, and one can envision a variety of conflicting inputs, which the controller must process in devising the proper control strategy.

5.2 Implementation

Attempts at inverse plant models using neural networks failed miserably, as might be expected with a thirteenth order nonlinear system model. It became clear that some variant of online learning would be required for training the controller. During the initial experimentation, it was noted that a neural network was able to very accurately predict the piston position with respect to time, and a form of feedback controller was devised based on this realization.

It was decided to train the controller to maintain a pump output of $4.8 \frac{L}{min}$ with a beat frequency of $1Hz$, unless the peripheral resistance appeared to drop. In that case, the controller was trained to beat more quickly (essentially as fast as the heart would fill), and to raise the systolic force a little, if it was appropriate.

The controller was given four inputs with which to make its control decision: current pump output; current systolic fraction; previous diastolic force, and; previous systolic force. This choice of control parameters means that there are no invasive sensors required; all control parameters can be measured from the VAD behaviour. This is an important consideration, because the body aggressively attacks any implanted sensors, and using pressure sensors, for example, is not practical.

The controller produced two outputs: a force adjustment for the base diastolic force, and; a force adjustment for the systolic increment.

5.2.1 Tapped Delay Lines

Neural networks are particularly good at tracking signals, or minimizing error, and the training was designed to take advantage of this effect. During training, the controller was presented with its previous control output through a tapped delay line, as illustrated in Figure 5.3.

Figure 5.3: Controller inputs using a tapped delay line

5.2.2 Pump Output Estimation Parameter

As the control parameters are updated on a beat-by-beat basis, the current pump output is estimated in a similar fashion, at the end of each beat. The current pump output can be predicted as in (5.1).

$$C_{out} = \frac{(V_{max} - V_{min})}{t_{beat}} \quad (5.1)$$

Given this, a normalized difference in the current pump output, can be defined as in (5.2), which allows a pump output between $3.6 \frac{L}{min}$ and $6 \frac{L}{min}$, with the nominal pump output being $4.8 \frac{L}{min}$.

$$\Delta_{cout} = \frac{1}{1200} \cdot \left(4800 - \frac{60 \cdot (V_{max} - V_{min})}{t_{beat}} \right) \quad (5.2)$$

5.2.3 Systolic Fraction Variation

In a similar way, one can also define a normalized variation in systolic fraction (the ratio between systolic time and diastolic time), as in (5.3), which also accounts for variations in beat length.

$$\Delta_{sys} = \frac{(\Delta_t |_{beat(nom)} - \Delta_t |_{sys.(nom)}) - \frac{t_{dia} - t_{sys}}{\Delta_t |_{beat(actual)}}}{0.3} \quad (5.3)$$

The variation in systolic fraction was selected as a control parameter because it is easily measurable and the fill rate of the VAD (or the ejection rate) should give some information on the current state of the circulatory system, especially when combined with a pump output indicator.

All inputs to the network are normalized to lie in the range $-1 \leq x \leq 1$, to speed training.

5.2.4 Network Topology

The network was designed with two hidden layers, to help it accommodate non-linearities in the system. As there were four desired inputs and two outputs, the network topology, shown in Figure 5.4, was chosen as $4 - 6 - 4 - 2$, providing a minimal number of neurons (to speed control calculation and training) while still allowing sufficient coverage of the state space. This results in fewer than expected neurons in the first hidden layer, but any failings due to lack of neurons should be readily apparent in the training output, as the learning error holds constant.

Figure 5.4: Selected network topology

The tapped delay lines provide a “data window” which eliminates the need for recurrent neurons in either hidden layer. Recurrent neurons serve to record the network history, but the tapped delay lines present the network with a t_{n-1} state, which can be used in conjunction with the current pump output and beat rate to determine the effectiveness of the current control strategy.

It may be wise to employ recurrent neurons in a future version of the controller, but problems arise with controller state determination: Unless the recurrent neurons are in a known state, one can never predict how their input will affect the network. This makes recurrent neurons very useful for extended time series prediction, when the network is started from a known initial state, but proving controller reliability with recurrent neurons may be more of a challenge. One cannot selectively “tune” the body’s circulatory parameters in order to achieve a known initial system state upon implantation of a ventricular assist device.

5.2.5 Training Strategy

The controller (while in “learning” mode) is trained on a beat-by-beat basis. That is, the controller makes a control decision and this control input is applied for the duration of one beat. After the beat is finished, an error signal is calculated based on the desired system output, and the neural network is trained for ten epochs, or ten iterations of the backpropagation algorithm. Only ten sets of training data are kept at any given time, to prevent over-training. Also, because the controller is initially started from steady state, the controller learns the steady state conditions more closely than the subsequent conditions, which should lead to a reduction in control error. (The controller has less time to “make a mistake” and throw the system off the initial conditions before it gets effective training data.)

The controller was trained for 10s of simulation time, for nominal systemic conditions, but it was found to train slowly. To speed learning, the initial conditions were “jittered”, or thrown off their nominal values, and the controller was trained while it attempted to restore proper system behaviour. The peripheral resistance was first increased and then decreased, and the controller was trained for 10s of simulation time in each case. After it had regained control of the pump output, training was stopped. This “early stopping” technique is designed to prevent over-training of the neural network.

Using this training strategy, the controller was found to over-learn the first condition it was presented with, and this lead to instabilities when the system was run at off-nominal conditions. To combat this problem, training time was increased disproportionately for the two phases of training. The controller was trained for 15s of simulation time for a decrease in peripheral resistance and then trained for 25s for an increase in peripheral resistance. This training strategy lead to an extremely stable controller that was still able to recognize variations in peripheral

resistance.

5.3 Controller Performance

There are numerous performance measures which can be used to evaluate a given controller. For the purposes of this work, the neural network controller will be judged against the standard full-fill, full-eject controller, with which it shares many characteristics.

5.3.1 Pump Output

Using the pump output (5.2) normalization, the controller pump output tends towards $4.8 \frac{L}{min}$. Under nominal systemic conditions, the trained controller holds the pump output (shown in Figure 5.5) near $4.8 \frac{L}{min}$ for the duration of the simulation. The “spikes” in the pump output are due to the estimation parameter, and small corrections made by the controller. If the graph were plotted on a larger scale, the peaks wouldn’t be noticeable, as they represent a very small percentage change.

Figure 5.5: Pump output under nominal systemic conditions

5.3.2 Beat Rate

For a full-fill, full-eject controller the beat rate is proportional to the pump output because the total blood volume is ejected with each beat. The VAD volume is specified as 80ml , which gives a beat rate of approximately 60bpm under nominal simulation conditions.

5.3.3 Pressure-Volume Loops

Pressure-volume plots of the VAD can be used to monitor integration error, and controller impact. With the VAD internal pressure plotted on the vertical axis, and VAD internal volume plotted on the horizontal axis, any variation in VAD volume at the beginning of systole or diastole is clearly seen, along with any internal average pressure increase.

The pressure volume loop for nominal simulation conditions, plotted in Figure 5.6, shows a slight systolic drift, due to a problem with initial VAD volume conditions, which corrects itself after approximately three beats.

Figure 5.6: Pressure-volume loop under nominal conditions

5.3.4 Changes in Peripheral Resistance

Changes in the body's peripheral resistance can be caused by exercise (as mentioned previously), external temperature changes, or a variety of other factors. The natural heart has an advantage over an artificial one because it has chemical receptors which help it in determining proper responses to changes in circulatory system parameters. The neural network was trained to maintain a $1Hz$ beat frequency *unless* the peripheral resistance decreased, in which case it was trained to beat as fast as it was able to fill. Obviously different control actions could have been selected; these seemed an appropriate test.

Increase in Peripheral Resistance

The controller response for a $2\times$ initial step increase in peripheral resistance is to increase the pumping force in an effort to maintain proper pump output, as illustrated in Figure 5.7. Over the course of a 20s simulation, the neural network controller manages to reduce the beat

Figure 5.7: Controller response after an increase in peripheral resistance

length to $1.1s$, where a standard full-fill full-eject controller beat length is $1.56s$. This represents

a difference of $1.3 \frac{L}{min}$ in pump output, or approximately $17bpm$, which is quite significant. The beat lengths are shown in Figure 5.8 and the pump output is shown in Figure 5.9.

Figure 5.8: Beat length after an increase in peripheral resistance

As the controller increases the pumping force in order to restore proper pump output, the maximum output cannula pressure (during systole) rises from $130mmHg$ to $145mmHg$, which is significant, but still within the acceptable range.

The peripheral flow, shown in Figure 5.11, is raised from a low of approximately $3.7 \frac{L}{min}$ to $4.2 \frac{L}{min}$ over the course of the simulation. The controller comes close to its goal of a $4.8 \frac{L}{min}$ pump output.

The pressure-volume loop for the VAD, plotted in Figure 5.12 shows a significant increase in systolic pressure over the first several beats.

Figure 5.9: Pump output after an increase in peripheral resistance

Figure 5.10: Output cannula pressure after an increase in peripheral resistance

Figure 5.11: Peripheral flow after an increase in peripheral resistance

Figure 5.12: Pressure-Volume loop for VAD after an increase in peripheral resistance

Decrease in Peripheral Resistance

A decrease in peripheral resistance is one way that the body signals a need for higher pump output. The controller response (shown in Figure 5.13) to a $2\times$ initial step decrease in peripheral resistance is to reduce the base and systolic forces very slightly (which aids filling), and then to maintain that force level throughout the course of the simulation. This control input

Figure 5.13: Controller response after a decrease in peripheral resistance

allows the neural-network to match the performance of a standard full-fill, full-eject controller, with a beat rate of approximately $0.85s$, and a pump output of $5.6 \frac{L}{min}$.

As the controller makes its force adjustments, the maximum pressure in the output cannula drops from $130mmHg$ to $115mmHg$. This pressure falls in the allowable range for systolic pressures, and the pressure difference between diastole and systole does not change, as can be seen on Figure 5.16.

The peripheral flow peaks at $5.9 \frac{L}{min}$, and settles at $5.3 \frac{L}{min}$ by the end of the simulation, slightly higher than the $4.7 \frac{L}{min}$ seen under nominal conditions. (Recall that one of the controller

Figure 5.14: Beat length after a decrease in peripheral resistance

Figure 5.15: Pump output after a decrease in peripheral resistance

Figure 5.16: Output cannula pressure after a decrease in peripheral resistance

design goals was to provide increased pump output when the peripheral resistance drops).

Figure 5.17: Peripheral flow after a decrease in peripheral resistance

The pressure-volume loop for the VAD, shown in Figure 5.18, indicates that the decrease in systolic pressure happens mainly over the first four beats, at which time the system settles into extremely predictable behaviour.

5.4 Controller Sensitivity

5.4.1 Initial Condition Variations

The controller and the system were tested for sensitivity to initial conditions. The system's "initial conditions" are described by the 1×13 vector of volumes, momenta, and piston position, and serve to fix the states of the various circulatory system elements upon startup.

Obviously one could vary the blood momentum or volume of each element in turn, and study how this affects the system, but after these transients had died out, the overall system

Figure 5.18: Pressure-Volume loop for VAD after a decrease in peripheral resistance

behaviours would be very similar. For this reason, two significant parameters were varied: the output cannula volume, and; the peripheral momentum. These were chosen because the output cannula is the first system element on the ejection side of the VAD, and because the peripheral momentum is directly related to the pump output, or the blood flow rate in the extremities.

Full-fill, full-eject controllers are more sensitive to initial conditions than are fixed rate controllers because a full-fill, full-eject controller's beat rate depends on the systemic pressures, but use of a fixed rate controller can mask volumetric dependence, and one must keep this in mind when analyzing simulation results.

The bond graph modeling method is energy-based; it accounts for the transfer of energy or power from one system element to another. This means a bond graph model will produce the minimum number of independent states necessary to fix the system in time, but this does not mean that the system has a clearly defined reference frame. This becomes readily apparent when a fixed rate controller is employed in the simulation, because any change in initial conditions produces a piston migration which effectively "creates" or "absorbs" volumes of blood. Note that the piston position, (4.3), is *not* an independent state.

By allowing the extra piston movement, a fixed rate controller (or a variant thereof) can restore nominal systemic conditions without any changes in systolic or diastolic force. In other words, changes in system blood volumes are reduced to transient disturbances.

In a full-fill, full-eject controller, on the other hand, piston motion is essentially a controller parameter, and is confined to a certain window.

Output Cannula Blood Volume Increase

When the output cannula blood volume is doubled on startup to 374ml (refer to Table 4.1 for the nominal initial conditions), the initial pressures in the output cannula, the aorta and the arterial system are very oscillatory, as can be seen in Figure 5.19, which illustrates the output cannula behaviour. (The aorta and the arteries behave in a similar fashion.) The major pressure oscillations die out after approximately 0.75s , because of the stiffness in the cannula, and the output cannula has smaller pressure oscillations than the aorta, owing once again to its higher compliance.

Figure 5.19: Pressure in the output cannula after an increase in output cannula blood volume

The preliminary beat lasts an amazing 3.24s, due to the high output cannula pressure and, because of the extra blood volume, the system pressure remains high throughout the rest of the simulation, even as the blood is dispersed more evenly throughout the circulatory elements. (This volume equalization happens mainly during the first beat.)

As might be expected, the extra blood eventually pools in the peripheral venous system, as seen in Figure 5.20, because of the very low compliance of the peripheral veins. Interestingly, the systolic pressures in the output cannula, aorta and arteries all return to near their nominal values, but the diastolic pressures remain much higher than usual, as can be seen in Figure 5.19. This dramatic increase in diastolic pressure changes the response of the output cannula

Figure 5.20: Peripheral venous volume after an increase in output cannula blood volume

and the aorta; the diastolic notch is no longer visible, and the pressure waveform more closely resembles that seen in the VAD itself. The total pressure differential (between systole and diastole) in the output cannula is reduced to 4 mmHg , a fraction of the 40 mmHg seen under nominal conditions. This is particularly bad.

To complicate matters, the pump output, plotted in Figure 5.21, is much lower than usual, and the controller immediately begins raising the systolic force in an attempt to increase the pump output by shortening the beat rate. This control input, plotted in Figure 5.22, increases

Figure 5.21: Pump output after an increase in output cannula blood volume

the ejection pressures even further, but over the course of the simulation they do not reach “critical” levels (which would require pressures of 175mmHg or higher.) After 30s the pump output has been increased to $4.8\frac{\text{L}}{\text{min}}$, and the peripheral flow rate has increased to $4.5\frac{\text{L}}{\text{min}}$. Obviously the system has not yet reached steady state.

Output Cannula Blood Volume Decrease

A decrease in systemic blood volume is perhaps more interesting from an application point of view, because patients tend to bleed during surgery, and (as mentioned previously) operation of the full-fill, full-eject controller is dependent on system pressures.

If the output cannula blood volume is halved on startup, representing a blood loss of 94ml , the pressures in the output cannula, aorta and arteries are once again very oscillatory, as

Figure 5.22: Controller response after an increase in output cannula blood volume

illustrated in Figure 5.23. These pressure oscillations die out after $0.4s$, or the length of the first beat.

Figure 5.23: Pressure in the output cannula after a decrease in output cannula blood volume

Once again, the variations in system blood volume are taken up by the peripheral veins, and the venous volume drops (shown in Figure 5.24) as the output cannula, aorta and arterial volumes return to near nominal levels.

The (initially) lower output cannula volume shortens the first few beats, which can be seen in Figure 5.24, as seven beats occur within the first $5s$ of simulation time. The controller compensates by increasing the diastolic force and the systolic increment, as shown in Figure 5.25. The controller lowers the systolic force increment for the remainder of the simulation, but begins increasing the diastolic force at $t = 18s$, which reduces the fill rate, and raises the beat length to $0.93s$. The controller decreases the systolic force at a fairly fixed rate, right up to the end of the simulation run. If the simulation is run for a longer period of time, the controller eventually reaches its goal of a $1s$ beat length.

Figure 5.24: Peripheral venous volume after a decrease in output cannula blood volume

Figure 5.25: Controller response after an decrease in output cannula blood volume

It is particularly telling that the controller correctly interprets the systemic conditions; it does not mistake the initially short beat length as a decrease in peripheral resistance. In other words, the systolic fraction has proven itself to be a good control input, and the controller is showing itself to be stable when faced with unknown circulatory conditions. When the controller detects a set of conditions it has not been trained for, it attempts to adjust the beat frequency to $1Hz$.

Peripheral Momentum Increase

The peripheral momentum is directly related to the peripheral blood flow rate, or the blood flow in the extremities. When the peripheral flow rate is doubled upon startup, there are no pressure oscillations in the output cannula, or the aorta. The pressure waveform does change slightly, but this may be owing to startup transients, rather than the increase in momentum. The output cannula pressure is plotted in Figure 5.26.

Figure 5.26: Output cannula pressure after an increase in peripheral momentum

A look at the peripheral venous volume (Figure 5.27) shows a very slight decrease over the course of the simulation. This excess volume is taken up in the arteries.

Figure 5.27: Peripheral venous volume after an increase in peripheral momentum

The initial beat length of $0.64s$ is much too short, as the high peripheral momentum decreases both the systolic and diastolic periods. The controller counters this by decreasing the systolic force increment (thus increasing the systolic fraction) and increases the beat length to $0.98s$ after five beats. At this point the controller begins restoring the control forces to their nominal levels and holds the beat frequency at $1Hz$ for the duration of the simulation. (The VAD forces are plotted in Figure 5.28.)

Figure 5.28: Controller response after an increase in peripheral momentum

A temporary increase in peripheral momentum is another systemic change which might be interpreted as a decrease in peripheral resistance, and the controller applies the correct control input yet again. After $30s$ the peripheral flow rate is $4.7 \frac{L}{min}$, and the pump output is $4.8 \frac{L}{min}$. Once again the system has not yet reached steady state.

Peripheral Momentum Decrease

When the peripheral momentum is decreased on startup, the pressure waveforms in the output cannula (plotted in Figure 5.29) and the aorta are distorted, but there are no noticeable

oscillations.

Figure 5.29: Output cannula pressure after a decrease in peripheral momentum

There is also no obvious change in any of the systemic volumes, unlike the slight volume shift seen when the peripheral momentum was increased. The initial beat length of $0.83s$ is too short, but the second beat lasts $1.2s$, which is too long. This happens as the extremely low initial peripheral flow rate is raised from $3.4 \frac{L}{min}$ to $4.8 \frac{L}{min}$ within the first few seconds of simulation time.

The controller makes a relatively large adjustment to the systolic force increment after the second beat because of the large overshoot of the pump output goal. The control inputs, plotted in Figure 5.30, drop the beat length to $1.02s$, and after the second beat an additional increase in the systolic force reduces the beat length even further, to $0.98s$. The beat length continues decreasing as the controller slowly restores the VAD forces to nominal levels, trying to achieve a $1s$ beat length, and after $30s$ of simulation time the beat length is $0.95s$.

Figure 5.30: Controller response after a decrease in peripheral momentum

This corresponds to a pump output of $5.1 \frac{L}{min}$, and a peripheral flow rate of $4.8 \frac{L}{min}$, both of which are higher than their nominal values. The controller is still (cautiously) reducing the systolic force increment at the end of the simulation, which may eventually restore the beat frequency to $1Hz$.

5.4.2 Parameter Variations

VAD Inertance Increase

It is easy to get a physical understanding of variations in VAD inertance: In this model, the VAD's pumping force is supplied by a piston, and changes in the VAD inertance are equivalent to changes in the piston mass. A change in piston inertance would directly affect the force required to accelerate the mass, or the required diastolic and systolic forces.

Increasing the piston inertia has a distinct effect on the maximum systolic pressure within the VAD itself, and a corresponding effect on the output valve flow rate, plotted in Figure 5.31. The VAD pressure waveform for an increase in piston inertance is shown in Figure 5.32. The differences in the pressure curve (namely, the overshoot and oscillations at the onset of systole) are accentuated as the piston inertia is increased even further.

A plot of the pressure-volume loop for the artificial heart, after an increase in VAD inertia (Figure 5.33) shows that the full-fill, full-eject controller variant maintains proper pump behaviour. (A fixed rate controller might accumulate blood in the heart, for example, because the larger piston inertia would affect the ejected volume.)

VAD Inertance Decrease

A decrease in VAD inertia has a much less pronounced effect on the model. The output and input valve flow rates do not change significantly, and the output cannula, aorta, arterial and venous pressures also remain near their nominal values. The pressure-volume loop for the VAD (Figure 5.34) shows normal behaviour, with no pressure oscillations after the output valve opens.

Figure 5.31: Output valve flow rate for an increase in VAD inertia

Figure 5.32: VAD Pressure for an increase in VAD inertia

Figure 5.33: Pressure-volume loop for VAD after an increase in VAD inertance

Figure 5.34: Pressure-volume loop for VAD after a decrease in VAD inertance

Output Cannula Inertance Increase

As outlined in (2.5), the inertance of a hydraulic element is dependent on its cross-sectional area, its length, and the fluid density. While the blood density is assumed constant, changes in cannula length or cross-sectional area might affect the cannula inertance.

The cannulae are made of woven Dacron, which is reinforced in the circumferential direction, and are resistant to changes in cross-sectional area, but are easily stretched lengthwise. The length of the cannula may vary from patient to patient, depending on the placement of the artificial heart.

A $2\times$ increase in output cannula inertia, which is effectively an increase in length, has little effect on the internal VAD pressures, or the flow rates through the valves. However, it has a noticeable effect on the aorta pressure response, as shown in Figure 5.35. As the output cannula inertance is increased, the initial pressure oscillations in the aorta also increase in magnitude.

Figure 5.35: Aorta pressure after and increase in output cannula inertance

These initial oscillations also show themselves in the arterial flow rate, shown in Figure 5.36, and the arterial flow rate is greatly increased over the remainder of the simulation. Peripheral and venous pressures and flow rates are not affected significantly, because the large peripheral inertia essentially decouples the venous side of the model from the arterial side.

Figure 5.36: Arterial flow rate after an increase output cannula inertance

A plot of the pressure-volume relationship for the VAD shows a slight decrease in VAD volume over the course of the simulation, as in Figure 5.37.

If the output cannula inertance is further increased, the pressure waveforms in the output cannula and the aorta change shape. The aorta pressure more closely resembles the arterial pressure wave, and the dicrotic notch in the output cannula pressure increases in magnitude. This changes the flow across the output valve during systole, but the oscillations are not large enough to trigger the valve, or induce valve flutter. The flow across the output valve for an extremely large ($16\times$) increase in output cannula inertance is plotted in Figure 5.38. Smaller increases yield similar (slightly more muted) features.

Figure 5.37: Pressure-volume loop for VAD after an increase in output cannula inertance

Figure 5.38: Output valve flow for a large increase in output cannula inertance

Output Cannula Inertance Decrease

If the output cannula inertance is decreased (which implies a shortening of the cannula) the pressure waveforms in the VAD and output cannula are not affected significantly, but the aorta experiences large initial pressure oscillations, as shown in Figure 5.39.

Figure 5.39: Aorta pressure after a decrease in output cannula inertance

The high frequency aorta oscillations appear because a decrease in output cannula inertance increases the natural frequency, and the system is underdamped.

Once again the arterial flow rate is affected by the change in output cannula compliance. The arterial model fails, as the flow rate over the course of the simulation is negative. (The arterial flow rate is shown in Figure 5.40). As explained previously, the arterial volume is the blood volume in the compliance element itself, not the actual blood volume in the arteries.

The large negative arterial flow rate represents a transfer of blood volume from the compliance element to the system, and given the average arterial flow rate of $1.5 \frac{L}{min}$, the total

systemic blood volume will be dramatically affected within a short period of time.

The aorta and peripheral flow rates remain near their nominal values.

Figure 5.40: Arterial flow rate after a decrease in output cannula inertance

The pressure-volume loop for the VAD, plotted in Figure 5.41 shows no change in heart volume.

Aorta Inertance Increase

Unlike the cannulae, which maintain a relatively constant cross-sectional area over the course of the pump cycle, the aorta diameter changes during normal pump operation. A $2\times$ increase in the aorta inertance has minimal effect on the VAD and output cannula pressures, but the aorta pressure is more oscillatory than normal, even as the simulation progresses. The oscillations, shown in Figure 5.42, are large during the initial beat, and although they decrease in magnitude over subsequent beats, they are still noticeable.

Figure 5.41: Pressure-volume loop for the VAD after a decrease in output cannula inertance

Figure 5.42: Aorta pressure after an increase in aorta inertance

There is minimal change to the output cannula, arterial, or venous flow rates, or the flow rates through the valves.

Aorta Inertance Decrease

A decrease in aorta inertance has little effect on the VAD or output cannula pressure, but produces initial pressure oscillations in the aorta itself. After the initial transients have died out, the aorta pressure is much smoother than the nominal case, as shown in Figure 5.43.

Figure 5.43: Aorta pressure after a decrease in aorta inertance

There is once again minimal change to the peripheral or venous flow rates, or the flow rates through the valves.

Peripheral Inertance Increase

An increase in peripheral inertance has a dramatic effect on the system. Internal pressures in the VAD stay near nominal levels, but the beat length is affected, as shown in Figure 5.44.

The second beat is longer than under nominal conditions, and the controller responds appropriately, reducing the length of subsequent beats.

Figure 5.44: VAD pressure after an increase in peripheral inertance

As can be seen in Figure 5.45, the pressure in the aorta is higher than nominal, although the waveform remains normal. The pressure in the output cannula exhibits a similar increase, as does the pressure in the arteries.

The arterial flow rate drops over the course of the simulation, indicating that the arterial model has once again failed. The arterial flow rate is plotted in Figure 5.46, and has dropped to a negative value after the first beat.

Both of the venous elements exhibit a pressure drop, but the venous flow resistance does not increase, and the venous model remains valid. The flow across the input valve is nominal, but the flow across the output valve is reduced, as shown in Figure 5.47. Note that the controller has restored the pump output to near nominal levels by the end of the 5s simulation.

Figure 5.45: Aorta pressure after an increase in peripheral inertance

Figure 5.46: Arterial flow rate after an increase in peripheral inertance

Figure 5.47: Output valve flow after an increase in peripheral inertance

Peripheral Inertance Decrease

Decreasing the peripheral inertance has a significant affect on system behaviour, as above. Once again the internal pressures in the VAD remain near nominal levels, but the beat length is affected, as shown in Figure 5.48. The first and second beats are much shorter than the nominal case, but the controller has restored the proper beat frequency by the end of the 5s simulation.

Figure 5.48: VAD pressure after a decrease in peripheral inertance

The pressures in the output cannula, aorta and arteries drop initially, but have returned to near nominal levels after approximately four beats. The output cannula pressure is plotted in Figure 5.49. The aorta behaves similarly, and the pressure decrease in the arteries is of the same magnitude, although the waveform is different (as it always is.)

When peripheral inertance is decreased the arterial model remains valid, as can be seen in Figure 5.50. The arterial flow rate increases slightly over the nominal condition.

Figure 5.49: Output cannula pressure after a decrease in peripheral inertia

Figure 5.50: Arterial flow rate after a decrease in peripheral inertance

The peripheral venous system experiences a pressure rise, shown in Figure 5.51, but the vena cava system does not exhibit a significant change in pressure. The venous flow resistance does not change, as the peripheral venous pressure remains higher than the vena cava pressure. The peripheral venous pressure is restored to near nominal levels by the end of the 5s simulation.

Figure 5.51: Peripheral venous pressure after an increase in peripheral inertance

Input Cannula Inertance Increase

As with the output cannula, the length of the input cannula may vary from patient to patient. An small increase in input cannula inertance has very minimal effect on the simulation, but a large increase produces pressure oscillations in the input cannula, shown in Figure 5.52, and oscillatory flow rates in the vena cava and peripheral venous system. (Although the flow rates in the combined venous system do not change dramatically, the venous blood volume oscillates between the two segments. The venous flow rates are plotted in Figure 5.53.)

Figure 5.52: Input cannula pressure after an increase in input cannula inertance

Figure 5.53: Venous flow rates after an increase in input cannula inertance

The input cannula pressure oscillations also produce oscillations in the input valve flow, but they are not severe enough to cause valve flutter.

Input Cannula Inertance Decrease

A decrease in input cannula inertance makes no observable changes to the system flow rates or pressures. The flow in the input cannula shows oscillations similar to the nominal condition, even for a large decrease in input cannula inertance. The flow rate through the input valve and the pressure waveform in the VAD again closely match the nominal values. The venous system also behaves as under nominal conditions.

Chapter 6

Conclusions and Recommendations

This thesis improved the simulation and the model, and implemented a neural network controller which was successful.

6.1 Conclusions

The new differentiable valve resistances are able to match the behaviour of the old threshold valves, and speed numerical integration of the system.

The new venous model is able to effectively model the vena cava collapse (and the accompanying increase in flow resistance), and its addition to the system model eliminates the pressure oscillations in the input cannula. Prior to this work, addition of a flow resistance in the input cannula was investigated, although there was no physical reason to include such an element. The new venous model removes the need for additional damping in the input cannula, and is more representative of the human circulatory system.

The semi-implicit extrapolation method offers a large performance improvement over the previous variable Runge-Kutta integration scheme, taking only 10.4s to integrate the previous state equations, something which took the ODE45 routine 1313s to complete.

The neural network variant of the full-fill, full-eject controller performs very well, and maintains proper cardiac output when faced with variations in systemic parameters. It correctly recognizes changes in peripheral resistance, and does not misinterpret variations in other peripheral parameters.

The effects of varying the diastolic force can now be studied because of the new venous model; if the force becomes too low, the venous model will collapse. The diastolic force could be made negative, to draw blood into the heart if necessary, something that was not feasible with the previous venous model.

The diastolic force and systolic force increment show themselves to be excellent control parameters for maintaining proper cardiac output and reasonable VAD filling.

6.2 Future Work

The present model of the circulatory system is a thorough treatment of the “left” side of the heart. The aorta and venous segments are nonlinear, and the remaining elements do not exhibit any nonlinearities. The next logical step for improving the system model would be the addition of the natural heart, and perhaps the addition of the pulmonary loop.

The addition of the natural heart would allow the venous parameters to be lowered to more reasonable values, as the right ventricle would no longer be lumped with the venous model. It would also allow investigation into synchronous modes of VAD operation, and would permit use of the “residual beating” system parameter, which is currently set to zero.

Addition of the pulmonary loop would allow studies of total artificial heart performance, while the current model only allows studies of ventricular assist devices.

Other organs could be modeled using compliance elements, to see what effect (if any) their addition had on the system.

The full-fill, full-eject controller is difficult to tune, because the switch from diastole to systole cannot be explicitly controlled. This results in a lower efficiency, because the systolic fraction is larger than necessary. A “hybrid” controller, which allowed a variable delay between the switch from diastole to systole would increase the VAD efficiency dramatically.

Fixed-rate controllers can have efficiencies of 45%, while a full-fill, full-eject controller has an efficiency of only 15% [40]. The addition of a delay parameter would move the full-fill, full-

eject controller closer to the fixed rate controller in terms of performance, while still maintaining the flexibility which makes full-fill, full-eject controllers so attractive.

The neural network controller could be tuned with other control goals in mind: in this work it was trained to maintain a constant cardiac output unless the peripheral resistance decreased; perhaps other physiological parameter changes are a more pressing concern from a circulatory perspective.

It would be interesting to compare the performance of a fuzzy logic controller using similar design goals with the neural network controller. Fuzzy logic is more conducive to this sort of controller design, but implementing it in hardware presents a challenge. Neural networks, on the other hand, are harder to tune, but easier to implement in hardware.

The neural network's damage tolerance could be tested by eliminating neurons at random. This would become an important consideration if the control system were ever implemented; redundancy and damage tolerance are important.

A study should be undertaken to determine what reasonable controller goals are, what the indicators should be, and what they mean from a physiological point of view.

References

- [1] Altman, P.L., Dittmer, D.S. (eds.) 1971. *Respiration and Circulation*. Bethesda, MD.: Federation of American Societies for Experimental Biology.
- [2] Bishop, C.M. 1995. *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press.
- [3] Chen, C.H. (ed.) 1996. *Fuzzy Logic and Neural Network Handbook*. New York: McGraw-Hill.
- [4] Cooley, D.G. (ed.) 1964. *Family Medical Guide*. New York: Better Homes and Gardens.
- [5] Croston, R.C., et al. 1973. *Computer Model of Cardiovascular Control System Responses to Exercise*. J. of Dynamic Systems, Measurement, and Control. **95**:301-307.
- [6] Davis, S.R. 1996. *Learn Java Now* Microsoft Press.
- [7] Elanayar, S., Shin, Y.C. 1994. *Radial Basis Function Neural Network for Approximation and Estimation of Nonlinear Stochastic Dynamic Systems*. IEEE Trans. on Neural Networks. **5(4)**:594-603.
- [8] Franklin, J.A., Selfridge, O.G. 1990. *Some New Directions for Adaptive Control Theory in Robotics*. Neural Networks for Control. 349-360.
- [9] Fu, M., et al. 1997. *Design of a DSP Controller for an Innovative Ventricular Assist System*. ASAIO Journal. **43**:M615-M619.
- [10] Fung, Y.C. 1984. *Biodynamics: Circulation*. New York: Springer-Verlag.
- [11] Ghista, D.N. 1990 *Clinical Cardiac Assessment, Interventions and Assist Technology*. Adv. Cardiovascular Phys. **7**:1-25.

- [12] Gray, Henry. 1995. *Anatomy: Descriptive and Surgical*. London: Magpie.
- [13] Hendry, P.J., et al. 1996. *Evolution of an Electrohydraulic Ventricular Assist Device Through In Vivo Testing*. ASAIO Journal. **42**:M350-M354.
- [14] Hendry, P., et al. 1999. *In Vivo Evaluation of an Intrathoracic Ventricular Assist Device*. ASAIO Journal. **45**:123-126.
- [15] Jang, J.-S.R., et al. 1997. *Neuro-fuzzy and Soft Control*. Upper Saddle River, NJ: Prentice Hall.
- [16] Johnson, K.E., et al. 1992. *Use of Total Artificial Hearts*. ASAIO Journal. **38**:M486-492.
- [17] Jordan, M.I. 1988. *Sequential Dependencies and Systems with Excess Degrees of Freedom*. COINS Technical Report 88-27.
- [18] Jorgensen, C.C., Schley, C. 1990. *A Neural Network Baseline Problem for Control of Aircraft Flare and Touchdown*. Neural Networks for Control. 403-426.
- [19] Karnopp, D.C., et al. 1990. *System Dynamics: A Unified Approach*. New York: John Wiley and Sons.
- [20] Katz, A.I., et al. 1969. *Flow Through a Collapsible Tube*. Biophysical Journal. **9**:1261-1279.
- [21] Kaufmann, R., et al. 1996. *The Implantable Fuzzy Controlled Helmholtz-Left Ventricular Assist Device: First In Vitro Testing*. Artificial Organs. **21(2)**:131-137.
- [22] Klute, G.K., et al. 1992. *An Optimal Controller for an Electric Ventricular-Assist Device: Theory, Implementation and Testing*. IEEE Trans. on Biomedical Eng. **39(4)**:394-403.
- [23] Ku, C., Lee, K.Y. 1995. *Diagonal Recurrent Neural Networks for Dynamic Systems Control*. IEEE Trans. on Neural Networks. **6(1)**:144-155.
- [24] Levin, A.U., Narendra, K.S. 1993. *Control of Nonlinear Dynamical Systems Using Neural Networks: Controllability and Stabilization*. IEEE Trans. on Neural Networks. **4(1)**:192-205.
- [25] Lu, Y.-Z. 1996. *Industrial Intelligent Control*. England: John Wiley and Sons.

-
- [26] Mel, B.W. 1990. *Vision-Based Robot Motion Planning*. Neural Networks for Control. 229-254.
- [27] Melchior, F.M., et al. 1992. *Mathematical modeling of human cardiovascular system for simulation of orthostatic response*. Am. J. Physiol., Heart Circ. Physiol. **31**:H1920-H1933.
- [28] Menon, V. 1997. *Fuzzy Logic Controller for an Artificial Heart*. Carleton University, ON.
- [29] Mukhopadhyay, S., Narendra, K.S. 1993. *Disturbance Rejection in Nonlinear Systems Using Neural Networks*. IEEE Trans. on Neural Networks. **4**(1):63-72.
- [30] Mussivand, T., et al. 1996. *A Remotely Controlled and Powered Artificial Heart Pump*. Artificial Organs. **20**(12):1314-1319.
- [31] Mussivand, T., et al. 1999. *Development of a Ventricular Assist Device for Out-of-Hospital Use*. J. of Heart and Lung Transplantation. **18**(2):166-171.
- [32] Peterfreund, N., Baram, Y. 1994. *Second-Order Bounds on the Domain of Attraction and the Rate of Convergence of Nonlinear Dynamical Systems and Neural Networks*. IEEE Trans. on Neural Networks. **5**(4):551-559.
- [33] Press, W.H., et al. 1992. *Numerical Recipes in C*. New York: Cambridge University Press.
- [34] Psaltis, D., et al. 1988. *A Multilayered Neural Network Controller*. IEEE Control Systems Magazine **8**:17-21.
- [35] Puskorius, G.V., Feldkamp, L.A. 1994. *Neurocontrol of Nonlinear Dynamical Systems with Kalman Filter Trained Recurrent Networks*. IEEE Trans. on Neural Networks. **5**(2):279-296.
- [36] Sarle, W.S., ed. 1997. *Neural Network FAQ, part 1 of 7: Introduction*. Periodic posting to the Usenet newsgroup comp.ai.neural-nets. URL: <ftp://ftp.sas.com/pub/neural/FAQ.html>
- [37] Srinivasan, V., et al. 1988. *Pattern Recognition and Feedback via Parallel Distributed Processing*. A.M. American Chemical Engineers. Washington, D.C.
- [38] Staib, W.E. 1995. *Neural Network Control System for Electric Arc Furnaces*. M.P.T. International. 2:58-61.

-
- [39] Steen, E.B., Montagu, Ashley. 1959. *Anatomy and Physiology*. New York: Barnes and Noble.
 - [40] Tan, S. 1992. *Development of a Dynamic Model of a Ventricular Assist Device for Investigation of Control Systems*. Carleton University, ON.
 - [41] Tortora, G.J., et al. 1984. *Principles of Anatomy and Physiology*. New York: Harper Row.
 - [42] White, D.A., Solge, D.A. (eds.) 1992. *Handbook of Intelligent Control*. New York: Multi-science Press.
 - [43] Yakowicz, S., Szidarovszky, F. 1989. *An Introduction to Numerical Computation*. New York: Macmillan.
 - [44] Miller, W.T., et al. 1990. *Neural Networks for Control*. Cambridge, MA: MIT Press.

Appendix A

Package ai.neural

<i>Package Contents</i>	<i>Page</i>
Classes	
Cluster	136
ConnectionGroup	138

A.1 Classes

A.1.1 Class **Cluster**

This is a doubly linked list of nodes.

Declaration

```
public class Cluster
extends mjc.util.Group
```

Constructors

- *Cluster*

```
public Cluster( )
```

- **Usage**

- * Creates an empty cluster of nodes.

Methods

- *add*

```
public void add( ai.neural.neuron.Node node )
```

- **Usage**

- * Adds a node to the cluster.

- **Parameters**

- * **node** - The node to be added to the cluster.

- *computeOutput*

```
public Matrix computeOutput( )
```

- **Usage**

- * Computes the output activations of all the neurons in the cluster.

- **Returns** - A matrix of neuron activations.

- *dimension*

public Matrix **dimension**()

- **Usage**

- * Returns the size of the cluster, that is, the number of neurons.

- **Returns** - The number of neurons in the cluster. (MxN)

- *getNumber*

public long **getNumber**()

- **Usage**

- * Returns the topological number of the layer.

- **Returns** - The topological number of the layer.

- *initialize*

public void **initialize**()

- **Usage**

- * Initializes all of the neurons in the cluster.

- *isInputLayer*

public boolean **isInputLayer**()

- **Usage**

- * Returns 'true' if this is an input cluster, and 'false' if it is not.

- **Returns** - 'True' if this is an input cluster, and 'false' if it is not.

- *setInputs*

public void **setInputs**(numerical.matrix.Vector **inputs**)

- **Usage**

- * Sets the input activations to the cluster.

- **Parameters**

- * **inputs** - A vector of input activations.

- *setNumber*

```
public void setNumber( long newnumber )
```

- **Usage**

- * Sets the topological number of the cluster.

- **Parameters**

- * **newnumber** - The new topological number of the cluster.

- *update*

```
public void update( )
```

- **Usage**

- * Updates all of the neurons in the cluster.

A.1.2 Class **ConnectionGroup**

This is a doubly linked list of connections.

Declaration

```
public class ConnectionGroup  
extends mjc.util.Group
```

Constructors

- *ConnectionGroup*

```
public ConnectionGroup( )
```

- **Usage**

- * Creates an empty connection group.

Methods

- *add*

```
public void add( ai.neural.Cluster clusterA, ai.neural.Cluster  
clusterB )
```

– **Usage**

- * Creates a full set of connections between two clusters.

– **Parameters**

- * `clusterA` - The source cluster.
 - * `clusterB` - The destination cluster.
-

- *add*

```
public void add( ai.neural.connection.Connection connection )
```

– **Usage**

- * Adds a connection to the connection group.

– **Parameters**

- * `connection` - The connection to be added.
-

- *add*

```
public void add( ai.neural.neuron.Node nodeA, ai.neural.neuron.Node  
nodeB )
```

– **Usage**

- * Adds a connection between two neurons to the connection group.

– **Parameters**

- * `nodeA` - The source node.
 - * `nodeB` - The destination node.
-

- *getNumber*

```
public long getNumber( )
```

– **Usage**

- * Returns the topological number of the connection group.

– **Returns** - The topological number of the connection group.

- *initialize*

```
public void initialize( )
```

– **Usage**

-
- * Initializes all of the connections in the connection group.
-

- *insertConnection*

```
public void insertConnection( ai.neural.connection.Connection  
connectionA, ai.neural.connection.Connection connectionB )
```

- **Usage**

- * Inserts a connection into the linked list.

- **Parameters**

- * **connectionA** - The connection to be inserted.
 - * **connectionB** - The connection which precedes the new connection.
-

- *prune*

```
public void prune( )
```

- **Usage**

- * Gets rid of redundant connections.
-

- *setNumber*

```
public void setNumber( long newnumber )
```

- **Usage**

- * Sets the topological order number of the connection group.

- **Parameters**

- * **newnumber** - The new topological order number.
-

- *update*

```
public void update( )
```

- **Usage**

- * Updates all of the connections in the connection group.

Appendix B

Package ai.neural.neuron

<i>Package Contents</i>	<i>Page</i>
Classes	
NodeComparator	142
Node	143
Context	148
Neuron	151
InputNode	155
OutputNode	156

B.1 Classes

B.1.1 Class **NodeComparator**

The node comparator allows comparison of two nodes.

Declaration

```
public class NodeComparator
extends java.lang.Object
implements mjc.util.Comparator, java.io.Serializable
```

Constructors

- *NodeComparator*
public **NodeComparator**()

Methods

- *equalTo*
public boolean **equalTo**(java.lang.Object **a**, java.lang.Object **b**)
 - **Usage**
 - * Checks to see if two nodes are equal.
 - **Parameters**
 - * **a** - The first node.
 - * **b** - The second node.
 - **Returns** - 'True' if 'a' and 'b' are references to the same object; 'false' otherwise.
- *greaterThan*
public boolean **greaterThan**(java.lang.Object **a**, java.lang.Object **b**)
 - **Usage**
 - * Compares the topological order of two nodes.
 - **Parameters**

- * a - The first node.
- * b - The second node.

– **Returns** - 'True' if 'a' has a higher topological number than 'b'; 'false' otherwise.

- *lessThan*

```
public boolean lessThan( java.lang.Object  a, java.lang.Object  b )
```

– **Usage**

- * Compares the topological order of two nodes.

– **Parameters**

- * a - The first node.
- * b - The second node.

– **Returns** - 'True' if 'a' has a lower topological number than 'b'; false otherwise.

- *typeCheck*

```
public void typeCheck( java.lang.Object  a )
```

– **Usage**

- * Performs type-checking on the input object.

– **Parameters**

- * a - The node to be tested.

– **Exceptions**

- * `java.lang.RuntimeException` - If the object is not an instance of `ai.neural.neuron.Node`.

B.1.2 Class **Node**

A node is a generic neural network element. This is an abstract class which can be extended for a hidden-layer neuron, an input node, or an output node.

Declaration

```
public abstract class Node
extends java.lang.Object
implements java.io.Serializable
```

Constructors

• *Node*

```
public Node( )
```

– **Usage**

- * Creates a generic neural network node.

Methods

• *accumulateError*

```
public void accumulateError( double delta, long connectionID )
```

– **Usage**

- * Accumulates an error increment for the node.

– **Parameters**

- * **delta** - The error increment.
- * **ID** - The corresponding outgoing connection ID.

• *accumulateInput*

```
public void accumulateInput( double input, long connectionID )
```

– **Usage**

- * Accumulates node input and applies the input function.

– **Parameters**

- * **input** - The input activation increment.
- * **neuronID** - The incoming connection ID.

• *computeError*

```
public double computeError( )
```

– **Usage**

- * Computes the output error for the node.

– **Returns** - The output error for the node.

- *computeOutput*

public double **computeOutput**()

- **Usage**

- * Computes the output activation for the node.

- **Returns** - The output activation for the node.

- *getID*

public long **getID**()

- **Usage**

- * Returns the unique ID number for the node.

- **Returns** - The unique node ID.

- *getLayerNumber*

public long **getLayerNumber**()

- **Usage**

- * Returns the current layer number for the node.

- **Returns** - The layer number.

- *getNumber*

public long **getNumber**()

- **Usage**

- * Returns the topological order number for the node.

- **Returns** - The topological number for the node.

- *getType*

public long **getType**()

- **Usage**

- * Returns the "type" of the node. Current types include:

- ai.neural.neuron.Node.CONTEXT
 - ai.neural.neuron.Node.HIDDEN
 - ai.neural.neuron.Node.INPUT

· ai.neural.neuron.Node.OUTPUT

– **Returns** - The node type.

- *initialize*

public abstract void **initialize**()

– **Usage**

* Initializes the node.

- *name*

public String **name**()

– **Usage**

* Returns a topological description of the node, including the topological order number, and the layer number.

– **Returns** - The topological description of the node.

- *registerConnection*

public void **registerConnection**(ai.neural.connection.Connection
connection, long **id**, boolean **incoming**)

– **Usage**

* Records an incoming or outgoing connection to the node.

– **Parameters**

* **id** - The ID of the connection.

* **incoming** - 'True' if this is an incoming connection, 'false' otherwise.

- *setDebugMode*

public void **setDebugMode**(boolean **yesno**)

– **Usage**

* Toggles debug mode for the node. Nodes in debug mode produce more verbose output.

– **Parameters**

* **yesno** - 'True' places the node in debug mode, 'false' does the opposite.

- *setID*

protected void **setID**(long **newID**)

- **Usage**

- * Changes the node's unique ID. Use with caution!

- **Parameters**

- * **newID** - The new unique ID.

- *setLayerNumber*

public void **setLayerNumber**(long **layernumber**)

- **Usage**

- * Changes the neuron's apparent layer number. This can affect the update order in some learning methods.

- **Parameters**

- * **layernumber** - The new layer number.

- *setNumber*

public void **setNumber**(long **newnumber**)

- **Usage**

- * Changes the topological number of the neuron.

- **Parameters**

- * **newnumber** - The new topological order number.

- *toString*

public abstract String **toString**()

- **Usage**

- * This abstract method must be overridden, and should provide a text description of the node's state.

- **Returns** - A text description of the node.

- *update*

public boolean **update**()

- Usage

- * Updates the output activation of the node. This method should probably be abstract.

- Returns - 'True' if successful, 'false' otherwise.**B.1.3 Class Context**

A context neuron is found in a recurrent network.

Declaration

```
public class Context
extends ai.neural.neuron.Node
```

Constructors

- *Context*

```
public Context( )
```

- Usage

- * Creates a generic context neuron.
-

- *Context*

```
public Context( ai.neural.neuron.functions.InputFunction
inputfunction, ai.neural.neuron.functions.TransferFunction
transferfunction, ai.neural.neuron.functions.OutputFunction
outputfunction )
```

- Usage

- * Creates a context neuron with the specified input, transfer, and output functions.

- Parameters

- * **inputfunction** - The new input function.
- * **transferfunction** - The new transfer function.
- * **outputfunction** - The new output function.

Methods

- *accumulateError*

```
public void accumulateError( double delta, long connectionID )
```

- **Usage**

- * Accumulates neuron error. This method is useful for backpropagation-type algorithms.

- **Parameters**

- * **delta** - The error increment.
 - * **connection** - The ID of the associated connection.
-

- *accumulateInput*

```
public void accumulateInput( double input, long connectionID )
```

- **Usage**

- * Accumulates input to the neuron as dictated by the input function.

- **Parameters**

- * **input** - The input increment.
 - * **connectionID** - The associated incoming connection.
-

- *computeError*

```
public double computeError( )
```

- **Usage**

- * Computes the error associated with this neuron.

- **Returns** - The total error associated with this neuron.

- *computeOutput*

```
public double computeOutput( )
```

- **Usage**

- * Computes the output activation for this neuron.

- **Returns** - The output activation.

- *computeOutput*

```
public Matrix computeOutput( numerical.matrix.Matrix inputs )
```

- **Usage**

- * Computes the output activation of a context neuron.

- **Parameters**

- * The - neuron inputs.

- **Returns** - The output activation (1x1 matrix).

- *initialize*

```
public void initialize( )
```

- **Usage**

- * Initializes the neuron and its transfer functions.

- *setInputFunction*

```
public void setInputFunction( ai.neural.neuron.functions.InputFunction  
inputfunction )
```

- **Usage**

- * Changes the neuron's input function.

- **Parameters**

- * **inputfunction** - The new input function.

- *setOutputFunction*

```
public void setOutputFunction(  
ai.neural.neuron.functions.OutputFunction outputfunction )
```

- **Usage**

- * Changes the neuron's output function.

- **Parameters**

- * **outputfunction** - The new output function.

- *setTransferFunction*

```
public void setTransferFunction(  
ai.neural.neuron.functions.TransferFunction transferfunction )
```

– **Usage**

- * Changes the neuron's transfer function.

– **Parameters**

- * `transferfunction` - The new transfer function.
-

- *toString*

```
public String toString( )
```

– **Usage**

- * Returns a descriptive string of the context neuron, including topological order, net input and output activation.

– **Returns** - A description of the neuron.

- *update*

```
public boolean update( )
```

– **Usage**

- * Updates the neuron's output activation.

– **Returns** - 'True' if the update was successful, 'false' otherwise.

B.1.4 Class Neuron

A generic hidden layer neuron accepts input, applies the appropriate transfer functions and produces an activation output.

Declaration

```
public class Neuron
extends ai.neural.neuron.Node
```

Constructors

- *Neuron*

```
public Neuron( )
```

– **Usage**

- * Creates a generic hidden layer neuron.

- *Neuron*

```
public Neuron( ai.neural.neuron.functions.InputFunction
inputfunction, ai.neural.neuron.functions.TransferFunction
transferfunction, ai.neural.neuron.functions.OutputFunction
outputfunction )
```

- **Usage**

- * Creates a hidden layer neuron with the specified transfer functions.

- **Parameters**

- * *inputfunction* - The input function.
- * *transferfunction* - The transfer function.
- * *outputfunction* - The output function.

Methods

- *accumulateError*

```
public void accumulateError( double delta, long connectionID )
```

- **Usage**

- * Accumulates output error for the neuron. This is useful for backpropagation-type learning algorithms.

- **Parameters**

- * *delta* - The error increment.
- * *connection* - The corresponding outgoing connection.

- *accumulateInput*

```
public void accumulateInput( double input, long connectionID )
```

- **Usage**

- * Accumulates input to the neuron by applying the input function.

- **Parameters**

- * *input* - The incremental input.
- * *connectionID* - The corresponding incoming connection.

- *computeError*

public double **computeError**()

- **Usage**

- * Computes the output error for the neuron. This method is useful for backpropagation-type learning algorithms.

- **Returns** - The output error.

- *computeOutput*

public double **computeOutput**()

- **Usage**

- * Computes the output activation of the neuron.

- **Returns** - The output activation.

- *computeOutput*

public Matrix **computeOutput**(numerical.matrix.Matrix **inputs**)

- **Usage**

- * Computes the activation output of the neuron. This is reserved for future use.

- **Returns** - The output activation – Currently NULL.

- *initialize*

public void **initialize**()

- **Usage**

- * Initializes the neuron and its transfer functions.

- *setInputFunction*

public void **setInputFunction**(ai.neural.neuron.functions.InputFunction **inputfunction**)

- **Usage**

- * Changes the neuron's input function.

- **Parameters**

* *inputfunction* - The new input function.

- *setOutputFunction*

```
public void setOutputFunction(  
ai.neural.neuron.functions.OutputFunction outputfunction )
```

- **Usage**

- * Changes the neuron's output function.

- **Parameters**

- * *outputfunction* - The new output function.

- *setTransferFunction*

```
public void setTransferFunction(  
ai.neural.neuron.functions.TransferFunction transferfunction )
```

- **Usage**

- * Changes the neuron's transfer function.

- **Parameters**

- * *transferfunction* - The new transfer function.

- *toString*

```
public String toString( )
```

- **Usage**

- * Returns a text description of the neuron, including topological number, layer number, net input and current activation.

- **Returns** - The text description of the neuron.

- *update*

```
public boolean update( )
```

- **Usage**

- * Updates the neuron's output activation.

- **Returns** - 'True' is the update was successful, 'false' otherwise.

B.1.5 Class **InputNode**

Input nodes have their activation set by the input pattern.

Declaration

```
public class InputNode
extends ai.neural.neuron.Neuron
```

Constructors

- *InputNode*
 public **InputNode**()
 - **Usage**
 - * Creates a generic input node.

Methods

- *computeOutput*
 public double **computeOutput**()
 - **Usage**
 - * Computes the activation output of the input node.
 - **Returns** - The activation output.
-
- *registerConnection*
 public void **registerConnection**(ai.neural.connection.Connection
 connection, long **id**, boolean **incoming**)
 - **Usage**
 - * Registers an outgoing connection to an input node.
 - **Parameters**
 - * **connection** - The connection to be added.
 - * **id** - The unique ID of the outgoing link.
 - * **incoming** - 'True' if this is an incoming link, 'false' otherwise.

– Exceptions

- * `java.lang.Exception` - If an incoming connection is attempted.

B.1.6 Class `OutputNode`

An output node's output activation forms part of the output pattern.

Declaration

```
public class OutputNode
extends ai.neural.neuron.Neuron
```

Constructors

- *OutputNode*

```
public OutputNode( )
```

– Usage

- * Creates a generic output node.

Methods

- *accumulateError*

```
public void accumulateError( double delta, long connectionID )
```

– Usage

- * Accumulates error for the output node.

– Parameters

- * `delta` - The error increment.
 - * `connectionID` - The outgoing connection ID (IGNORED!)
-

- *accumulateInput*

```
public void accumulateInput( double input, long connectionID )
```

– Usage

- * Accumulates input for the output node, and applies the node's input function.

- Parameters

- * `input` - The input increment.
- * `neuronID` - The unique ID of the incoming connection.

• `registerConnection`

```
public void registerConnection( ai.neural.connection.Connection  
connection, long id, boolean incoming )
```

- Usage

- * Records a connection to the node. Output nodes only accept incoming connections.

- Parameters

- * `connection` - The new connection.
- * `id` - The unique ID of the connection.
- * `incoming` - 'True' if this is an incoming connection, 'false' otherwise.

- Exceptions

- * `java.lang.Exception` - If an outgoing connection is attempted.

Appendix C

Package ai.neural.neuron.functions

<i>Package Contents</i>	<i>Page</i>
Interfaces	
OutputFunction	159
TransferFunction	159
InputFunction	161
Classes	
LogSig	163
Linear	165
TanSig	167
Product	169
Sum	170

C.1 Interfaces

C.1.1 Interface **OutputFunction**

The output function takes the unit activation and produces an output activation.

Declaration

```
public interface OutputFunction
```

Methods

- *computeValue*

```
public double computeValue( double value )
```

- **Usage**

- * Returns the output of the function.

- **Parameters**

- * value - The unit activation.

- **Returns** - The output activation.

- *idString*

```
public String idString( )
```

- **Usage**

- * Returns an identifier string for the function.

- **Returns** - The text identifier.

- *initialize*

```
public void initialize( )
```

- **Usage**

- * Initializes the function.

C.1.2 Interface **TransferFunction**

The transfer function takes the input to the unit and produces an activation.

Declaration

```
public interface TransferFunction
```

Methods

• *computeDerivative*

```
public double computeDerivative( double value )
```

– **Usage**

* Returns the function's derivative at the specified input.

– **Parameters**

* **value** - The independent variable.

– **Returns** - The function's derivative.

• *computeValue*

```
public double computeValue( double value )
```

– **Usage**

* Computes the output of the function.

– **Parameters**

* **value** - The independent variable.

• *getBias*

```
public double getBias( )
```

– **Usage**

* Returns the function's bias.

– **Returns** - The bias.

• *idString*

```
public String idString( )
```

– **Usage**

* Returns a text description of the function

– **Returns** - The text description.

- *initialize*

```
public void initialize( java.util.Random random )
```

– **Usage**

* Initializes the function with the given random number generator.

– **Parameters**

* **random** - The random number generator.

- *setBias*

```
public void setBias( double newbias )
```

– **Usage**

* Sets the function's bias.

– **Parameters**

* **newbias** - The new bias.

- *updateBias*

```
public void updateBias( double delta )
```

– **Usage**

* Updates the function's bias.

– **Parameters**

* **delta** - The incremental bias change.

C.1.3 Interface **InputFunction**

The input function takes a matrix of inputs and reduces it to a double. Examples: sum, product.

Declaration

```
public interface InputFunction
```

Methods

- *accumulateInput*

```
public void accumulateInput( double input )
```

- **Usage**

- * Accumulates input increments.

- **Parameters**

- * **input** - The input delta.

- *computeValue*

```
public double computeValue( )
```

- **Usage**

- * Returns the output of the function.

- **Returns** - The output.

- *getNetInput*

```
public double getNetInput( )
```

- **Usage**

- * Returns the net input to this function.

- **Returns** - The net input.

- *idString*

```
public String idString( )
```

- **Usage**

- * Returns an identifier string for the function.

- **Returns** - The text identifier.

- *initialize*

```
public void initialize( )
```

- **Usage**

- * Initializes the function.

C.2 Classes

C.2.1 Class **LogSig**

The logsig function can be used as a transfer function. It produces bounded output.

Declaration

```
public class LogSig
extends java.lang.Object
implements TransferFunction, java.io.Serializable
```

Constructors

- *LogSig*

```
public LogSig( )
```

- **Usage**

- * Creates a generic logsig transfer function.

Methods

- *computeDerivative*

```
public double computeDerivative( double value )
```

- **Usage**

- * Returns the function's derivative at the specified input.

- **Parameters**

- * **value** - The independent variable.

- **Returns** - The function's derivative.

- *computeValue*

```
public double computeValue( double value )
```

- **Usage**

- * Computes the output of the function.

– **Parameters**

* value - The independent variable.

- *getBias*

public double **getBias**()

– **Usage**

* Returns the function's bias.

– **Returns** - The bias.

- *idString*

public String **idString**()

– **Usage**

* Returns "LogSig", a text description of the function.

– **Returns** - The text description.

- *initialize*

public void **initialize**(java.util.Random **random**)

– **Usage**

* Initializes the function with the given random number generator.

– **Parameters**

* random - The random number generator.

- *setBias*

public void **setBias**(double **newbias**)

– **Usage**

* Sets the function's bias.

– **Parameters**

* newbias - The new bias.

- *updateBias*

public void **updateBias**(double **delta**)

- **Usage**

- * Updates the function's bias.

- **Parameters**

- * `delta` - The incremental bias change.

C.2.2 Class **Linear**

The linear function can be used as a transfer function or an output function. It does not produce bounded output.

Declaration

```
public class Linear
extends java.lang.Object
implements TransferFunction, OutputFunction, java.io.Serializable
```

Constructors

- *Linear*

```
public Linear( )
```

Methods

- *computeDerivative*

```
public double computeDerivative( double value )
```

- **Usage**

- * Returns the function's derivative at the specified input.

- **Parameters**

- * `value` - The independent variable.

- **Returns** - The function's derivative.

- *computeValue*

```
public double computeValue( double value )
```

– **Usage**

* Computes the output of the function.

– **Returns** - The output.

• *getBias*

public double **getBias**()

– **Usage**

* Returns the bias for the function.

– **Returns** - The bias.

• *idString*

public String **idString**()

– **Usage**

* Returns "Linear", a text description of the function.

– **Returns** - The text description.

• *initialize*

public void **initialize**()

– **Usage**

* Initializes the function.

• *initialize*

public void **initialize**(java.util.Random **random**)

– **Usage**

* Initializes the function with the given random number generator.

– **Parameters**

* **random** - The random number generator.

• *setBias*

public void **setBias**(double **newBias**)

– **Usage**

* Sets the function's bias.

– **Parameters**

* `newbias` - The new bias.

• *updateBias*

```
public void updateBias( double delta )
```

– **Usage**

* Updates the function's bias.

– **Parameters**

* `delta` - The incremental bias change.

C.2.3 Class **TanSig**

The tansig function can be used as a transfer function. It produces bounded output.

Declaration

```
public class TanSig
extends java.lang.Object
implements TransferFunction, java.io.Serializable
```

Constructors

• *TanSig*

```
public TanSig( )
```

– **Usage**

* Creates a generic tansig transfer function.

Methods

• *computeDerivative*

```
public double computeDerivative( double value )
```

– **Usage**

* Returns the function's derivative at the specified input.

– **Parameters**

* value - The independent variable.

– **Returns** - The function's derivative.

- *computeValue*

```
public double computeValue( double value )
```

– **Usage**

* Computes the output of the function.

– **Parameters**

* value - The independent variable.

- *getBias*

```
public double getBias( )
```

– **Usage**

* Returns the function's bias.

– **Returns** - The bias.

- *idString*

```
public String idString( )
```

– **Usage**

* Returns "TanSig", a text description of the function.

– **Returns** - The text description.

- *initialize*

```
public void initialize( java.util.Random random )
```

– **Usage**

* Initializes the function with the given random number generator.

– **Parameters**

* random - The random number generator.

- *setBias*

```
public void setBias( double  newbias )
```

- **Usage**

- * Sets the function's bias.

- **Parameters**

- * **newbias** - The new bias.

- *updateBias*

```
public void updateBias( double  delta )
```

- **Usage**

- * Updates the function's bias.

- **Parameters**

- * **delta** - The incremental bias change.

C.2.4 Class **Product**

The product function is an input function. It accumulates the inputs by multiplying them together.

Declaration

```
public class Product
extends java.lang.Object
implements InputFunction, java.io.Serializable
```

Constructors

- *Product*

```
public Product( )
```

Methods

- *accumulateInput*

```
public void accumulateInput( double  input )
```

- **Usage**

- * Accumulates an input increment.

- **Parameters**

- * `input` - The input delta.

- *computeValue*

```
public double computeValue( )
```

- **Usage**

- * Returns the output of the function.

- **Returns** - The output of the function.

- *getNetInput*

```
public double getNetInput( )
```

- **Usage**

- * Returns the net input to the function.

- **Returns** - The net input.

- *idString*

```
public String idString( )
```

- **Usage**

- * Returns "Product", a text description of the function.

- **Returns** - The text description.

- *initialize*

```
public void initialize( )
```

- **Usage**

- * Initializes the function.

C.2.5 Class **Sum**

The sum function is an input function. It accumulates inputs by adding them together.

Declaration

```
public class Sum
extends java.lang.Object
implements InputFunction, java.io.Serializable
```

Constructors

- *Sum*
public **Sum**()

Methods

- *accumulateInput*
public void **accumulateInput**(double **input**)
 - **Usage**
 - * Accumulates an input increment.
 - **Parameters**
 - * **input** - The input delta.

- *computeValue*
public double **computeValue**()
 - **Usage**
 - * Returns the output of the function.
 - **Returns** - The output of the function.

- *getNetInput*
public double **getNetInput**()
 - **Usage**
 - * Returns the net input to the function.
 - **Returns** - The net input.

- *idString*

```
public String idString( )
```

- **Usage**

- * Returns "Sum", a text description of the function.

- **Returns** - The text description.

- *initialize*

```
public void initialize( )
```

- **Usage**

- * Initializes the function.

Appendix D

Package ai.neural.connection

<i>Package Contents</i>	<i>Page</i>
Classes	
ConnectionComparator	174
Connection	175

D.1 Classes

D.1.1 Class **ConnectionComparator**

Compares two connections.

Declaration

```
public class ConnectionComparator
extends java.lang.Object
implements mjc.util.Comparator, java.io.Serializable
```

Constructors

- *ConnectionComparator*
public **ConnectionComparator**()

Methods

- *equalTo*
public boolean **equalTo**(java.lang.Object **a**, java.lang.Object **b**)
 - **Usage**
 - * Checks to see if two connections are equal.
 - **Parameters**
 - * **a** - The first connection.
 - * **b** - The second connection.
 - **Returns** - 'True' if 'a' and 'b' are references to the same object; false otherwise.
- *greaterThan*
public boolean **greaterThan**(java.lang.Object **a**, java.lang.Object **b**)
 - **Usage**
 - * Compares the topological order of two connections.
 - **Parameters**

- * a - The first connection.
 - * b - The second connection.
 - **Returns** - 'True' if 'a' has a higher topological number than 'b'; 'false' otherwise.
-

- *lessThan*

```
public boolean lessThan( java.lang.Object a, java.lang.Object b )
```

- **Usage**

- * Compares the topological order of two connections.

- **Parameters**

- * a - The first connection.
 - * b - The second connection.

- **Returns** - 'True' if 'a' has a lower topological number than 'b'; 'false' otherwise.
-

- *typeCheck*

```
public void typeCheck( java.lang.Object a )
```

- **Usage**

- * Performs type-checking on the input object.

- **Parameters**

- * a - The object to be tested.

- **Exceptions**

- * `java.lang.RuntimeException` - If the object is not an instance of `ai.neural.connection.Connection`.

D.1.2 Class **Connection**

A connection accepts input from one neuron, applies an optional transfer function, and transfers the activation to another neuron.

Declaration

```
public class Connection  
extends java.lang.Object  
implements java.io.Serializable
```

Constructors

- *Connection*

```
public Connection( ai.neural.neuron.Node nodeA,  
ai.neural.neuron.Node nodeB )
```

- **Usage**

- * Creates a new connection between two neurons.

- **Parameters**

- * neuronA - The source neuron.

- * neuronB - The target neuron.

Methods

- *getDestination*

```
public Node getDestination( )
```

- **Usage**

- * Returns a reference to the target neuron for this connection.

- **Returns** - The target neuron.

- *getID*

```
public long getID( )
```

- **Usage**

- * Returns the unique ID of this connection.

- **Returns** - The randomly-generated unique ID.

- *getPrevWeight*

```
public double getPrevWeight( )
```

- **Usage**

- * Returns the weight of this connection the previous timestep. Weights are normally only adjusted during training.

- **Returns** - The weight of the connection the previous timestep.

- *getSource*

public Node **getSource**()

- **Usage**

- * Returns a reference to the source neuron for this connection.

- **Returns** - The source neuron.

- *getWeight*

public double **getWeight**()

- **Usage**

- * Returns the current weight of the connection.

- **Returns** - The current weight of the connection.

- *initialize*

public void **initialize**()

- **Usage**

- * Initializes the connection..

- *setDebugMode*

public void **setDebugMode**(boolean **yesno**)

- **Usage**

- * Changes the debug mode for the connection. Debug mode affects the update of the connection – When 'debugMode' is true, connection updates are verbose.

- **Parameters**

- * yesno - Should the connection be set to debug mode?

- *setTransferFunction*

public void **setTransferFunction**(
ai.neural.neuron.functions.TransferFunction **transferfunction**)

- **Usage**

* Sets the connection's transfer function.

– **Parameters**

* **transferfunction** - The new transfer function for the connection.

• *setType*

```
public void setType( int newtype )
```

– **Usage**

* Sets the connection type to be 'NORMAL', 'DELAYED' or 'RECURRENT'.

– **Parameters**

* **newtype** - The new connection type.

• *toString*

```
public String toString( )
```

– **Usage**

* Returns a descriptive string for this connection.

– **Returns** - A description of the connection, including the source and target neuron IDs.

• *update*

```
public boolean update( )
```

– **Usage**

* Updates the connection by querying the source neuron's activation and transferring it to the target neuron.

– **Returns** - 'True' if successful and 'false' if not.

• *updateWeight*

```
public void updateWeight( double deltaweight )
```

– **Usage**

* Updates the connection's weight. This method is normally invoked by the training algorithm.

– **Parameters**

* **newweight** - The new connection weight.

Appendix E

Package ai.neural.network

<i>Package Contents</i>	<i>Page</i>
Classes	
Network	180
NetworkComparator	183
NetworkTimer	185

E.1 Classes

E.1.1 Class **Network**

This is a doubly-linked list of connection groups and clusters.

Declaration

```
public class Network
extends java.lang.Object
implements java.io.Serializable
```

Constructors

- *Network*
 public **Network**()
 - **Usage**
 - * Creates an empty network.

Methods

- *addCluster*
 public void **addCluster**(ai.neural.Cluster **cluster**)
 - **Usage**
 - * Adds a cluster to the network.
 - **Parameters**
 - * The - cluster to be added.
 - **Exceptions**
 - * java.lang.Exception - If there's a problem adding the cluster.
-
- *addConnection*
 public void **addConnection**(ai.neural.connection.Connection **connection**)

- **Usage**

- * Adds a connection to the network.

- **Parameters**

- * `connection` - The connection to be added.

- **Exceptions**

- * `java.lang.Exception` - If there's a problem adding the connection.

- *addConnectionGroup*

```
public void addConnectionGroup( ai.neural.ConnectionGroup  
connectiongroup )
```

- **Usage**

- * Adds a connection group to the network.

- **Parameters**

- * `connectiongroup` - The connection group to be added.

- **Exceptions**

- * `java.lang.Exception` - If there's a problem adding the connection group.

- *getName*

```
public String getName( )
```

- **Usage**

- * Returns the name of the network.

- **Returns** - The network name.

- *initialize*

```
public void initialize( )
```

- **Usage**

- * Initializes all of the connections and nodes in the network.

- *load*

```
public static Network load( java.io.FileInputStream in )
```

- **Usage**

* Loads a network from a disk file.

- **Parameters**

* *in* - The handle to the input file stream.

- **Returns** - The network.

• *setName*

```
public void setName( java.lang.String newname )
```

- **Usage**

* Changes the network's name.

- **Parameters**

* *newname* - The new name for the network.

• *simulate*

```
public Matrix simulate( numerical.matrix.Matrix inputs )
```

- **Usage**

* Computes the network outputs given the inputs.

- **Parameters**

* *inputs* - An MxN matrix of inputs, with each row corresponding to an input vector.

- **Returns** - An MxN matrix of outputs for the given inputs.

• *toString*

```
public String toString( )
```

- **Usage**

* Returns the network topology in (semi) human-readable form.

- **Returns** - The network topology as a string.

• *updateLists*

```
public void updateLists( )
```

- **Usage**

- * Updates the internal lists to match the clusters and groups. This method enumerates all of the network nodes and connections.

- *validate*

```
public Matrix validate( numerical.matrix.Matrix input,  
numerical.matrix.Matrix output )
```

- **Usage**

- * Computes the validation error for the network.

- **Parameters**

- * *input* - The network inputs.
 - * *output* - The desired network outputs.

- **Returns** - The SSE for each set of inputs.

E.1.2 Class **NetworkComparator**

The network comparator allows comparison of two networks.

Declaration

```
public class NetworkComparator  
extends java.lang.Object  
implements mjc.util.Comparator, java.io.Serializable
```

Constructors

- *NetworkComparator*

```
public NetworkComparator( )
```

Methods

- *equalTo*

```
public boolean equalTo( java.lang.Object a, java.lang.Object b )
```

- **Usage**

- * Checks to see if two networks are equal.

– **Parameters**

- * a - The first network.
- * b - The second network.

– **Returns** - 'True' if 'a' and 'b' are references to the same object; 'false' otherwise.

- *greaterThan*

```
public boolean greaterThan( java.lang.Object a, java.lang.Object b )
```

– **Usage**

- * Compares the topological order of two networks.

– **Parameters**

- * a - The first network.
- * b - The second network.

– **Returns** - 'True' if 'a' has a higher topological number than 'b'; false otherwise.

- *lessThan*

```
public boolean lessThan( java.lang.Object a, java.lang.Object b )
```

– **Usage**

- * Compares the topological order of two networks.

– **Parameters**

- * a - The first network.
- * b - The second network.

– **Returns** - 'True' if 'a' has a lower topological number than 'b'; false otherwise.

- *typeCheck*

```
public void typeCheck( java.lang.Object a )
```

– **Usage**

- * Performs type-checking on the input object.

– **Parameters**

- * a - The object to be tested.

– **Returns** - 'True' if 'a' is an instance of `ai.neural.network.Network`.

E.1.3 Class **NetworkTimer**

The network timer provides a global clock signal, which allows asynchronous node updates, and time-dependent transfer functions

Declaration

```
public class NetworkTimer
extends java.lang.Object
implements java.lang.Runnable, java.io.Serializable
```

Constructors

- *NetworkTimer*

```
public NetworkTimer( )
```

- **Usage**

- * Creates a generic network timer.

Methods

- *run*

```
public void run( )
```

- **Usage**

- * The timer method, which is run in a thread.

Appendix F

Package ai.neural.network.functions

<i>Package Contents</i>	<i>Page</i>
Classes	
TrainingFunction	187
Backpropagation	188
BackpropagationMomentum	190

F.1 Classes

F.1.1 Class **TrainingFunction**

The generic network training algorithm. All training algorithms should extend this class.

Declaration

```
public class TrainingFunction
extends java.lang.Object
```

Constructors

- *TrainingFunction*
public **TrainingFunction**()

Methods

- *setBatch*
public void **setBatch**(boolean **yesno**)
 - **Usage**
 - * Turns batch learning on or off.
 - **Parameters**
 - * **yesno** - 'True' turns batch learning on, while 'false' turns it off.
 - *setErrorTolerance*
public void **setErrorTolerance**(double **tolerance**)
 - **Usage**
 - * Sets the training algorithm's error tolerance. Network output errors which are smaller than this error tolerance are taken to be zero.
 - **Parameters**
 - * **tolerance** - The new error tolerance.
-

- *setMaxEpochs*

```
public void setMaxEpochs( int epochs )
```

- **Usage**

- * Sets the maximum number of network training epochs.

- **Parameters**

- * epochs - The new number of epochs.

- *setNetwork*

```
public void setNetwork( ai.neural.network.Network newnetwork )
```

- **Usage**

- * Sets the network to be trained.

- **Parameters**

- * newnetwork - The network to apply the training algorithm to.

- *train*

```
public void train( numerical.matrix.Matrix input,  
numerical.matrix.Matrix output )
```

- **Usage**

- * This is a generic training method. It does nothing, and *must* be overridden by a coded training algorithm.

- **Parameters**

- * input - The network inputs.

- * output - The desired network outputs.

- **Exceptions**

- * java.lang.Exception - Always. You must override this method.

F.1.2 Class **Backpropagation**

This is the vanilla backpropagation learning algorithm.

Declaration

```
public class Backpropagation  
extends ai.neural.network.functions.TrainingFunction
```


Constructors

- *Backpropagation*

public **Backpropagation**()

Methods

- *getLearningRate*

public double **getLearningRate**()

- **Usage**

- * Returns the current backpropagation learning rate.

- **Returns** - The current learning rate.

- *setDebugMode*

public void **setDebugMode**(boolean **yesno**)

- **Usage**

- * Changes the debug level for the learning algorithm. If the routine is run in debug mode it generates more verbose (somewhat overwhelming) output.

- **Parameters**

- * **yesno** - 'True' to turn on debugging and 'false' to turn it off.

- *setLearningRate*

public void **setLearningRate**(double **newlearningrate**)

- **Usage**

- * Sets the backpropagation learning rate.

- **Parameters**

- * **newlr** - The new learning rate.

- *train*

public void **train**(numerical.matrix.Matrix **input**,
numerical.matrix.Matrix **output**)

- Usage

- * Trains the network using the vanilla backpropagation algorithm.

- Parameters

- * **input** - The network inputs.
- * **output** - The desired network outputs.

F.1.3 Class BackpropagationMomentum

Implements backpropagation learning with momentum.

Declaration

```
public class BackpropagationMomentum
extends ai.neural.network.functions.Backpropagation
```

Constructors

- *BackpropagationMomentum*
public **BackpropagationMomentum**()

Methods

- *getMomentum*
public double **getMomentum**()
 - Usage**
 - * Returns the current momentum factor.
 - Returns** - The current momentum.
-
- *setMomentum*
public void **setMomentum**(double **newmomentum**)
 - Usage**
 - * Changes the current network learning momentum.
 - Parameters**

* **newmomentum** - The new momentum factor.

- *train*

```
public void train( numerical.matrix.Matrix input,  
numerical.matrix.Matrix output )
```

- **Usage**

- * Trains the network using backpropagation with momentum.

- **Parameters**

- * **input** - The network inputs.

- * **output** - The desired network outputs.

Appendix G

Package numerical.matrix

<i>Package Contents</i>	<i>Page</i>
Classes	
Matrix	193
Vector	211

G.1 Classes

G.1.1 Class **Matrix**

This is a matrix class.

Declaration

```
public class Matrix
extends java.lang.Object
```

Constructors

- *Matrix*

```
public Matrix( )
```

- **Usage**

- * Creates a 1x1 matrix and initializes it to 0.0

- *Matrix*

```
public Matrix( double [] matrix )
```

- **Usage**

- * Creates a new matrix and fills it with values from an array.

- **Parameters**

- * **matrix** - The array of doubles

- *Matrix*

```
public Matrix( int rows, int cols )
```

- **Usage**

- * Creates a new matrix of the specified size and initializes all its values to 0.0

- **Parameters**

- * **rows** - Number of rows.

- * **cols** - Number of columns.

- *Matrix*

```
public Matrix( java.lang.String string )
```

- **Usage**

- * Creates a matrix from the delimited string. The string must be delimited as follows:

- [- Begin matrix
 -] - End matrix
 - ; - End a row
 - , - Separate values (optional)
 - e.g. Matrix("[1 2 3; 4 5 6; 7 8 9;]");

- **Parameters**

- * The - string to parse.

- *Matrix*

```
public Matrix( java.lang.String string, int size )
```

- **Usage**

- * Creates one of several types of square matrices specified by an identifier string:

- IDENTITY - Creates an identity matrix
 - UPPERTRI - Creates an upper triangular matrix
 - LOWERTRI - Creates a lower triangular matrix

- **Parameters**

- * **string** - The identifier string.
 - * **size** - The size of the matrix.

Methods

- *addCols*

```
public void addCols( int cols )
```

- **Usage**

- * Adds the specified number of columns to the matrix.

- **Parameters**

- * **cols** - The number of columns to add.

- *addRows*

public void **addRows**(int rows)

- **Usage**

- * Adds the specified number of rows to the matrix.

- **Parameters**

- * rows - The number of rows to add.

- *asVector*

public Vector **asVector**(boolean rowordered)

- **Usage**

- * Returns the matrix as a vector in row or column order.

- **Parameters**

- * rowordered - 'True' returns the matrix in row order, 'false' in column order.

- **Returns** - The matrix as a vector.

- *cnorm*

public double **cnorm**()

- **Usage**

- * Returns the max column norm.

- **Returns** - Maximum of the column sums.

- *colSum*

public Matrix **colSum**()

- **Usage**

- * Sums the entries in each column.

- **Returns** - Matrix of column sums.

- *cond*

public double **cond**()

- **Usage**

- * Returns the condition number for the matrix.

- **Returns** - The condition number for the matrix.

- **Exceptions**

- * `java.lang.Exception` - If the SVD fails.

- *det*

```
public double det( )
```

- **Usage**

- * Returns the determinant of the matrix.

- **Returns** - The determinant of the matrix.

- **Exceptions**

- * `java.lang.Exception` - If the matrix isn't square.

- * `java.lang.Exception` - If the LU-decomposition fails.

- *dividedby*

```
public static Matrix dividedby( double scalar, numerical.matrix.Matrix  
matrix )
```

- **Usage**

- * Divides a matrix by a scalar and returns a matrix.

- **Parameters**

- * `scalar` - The scaling factor.

- * `matrix` - The matrix to be scaled.

- **Returns** - The matrix product

- *dividedby*

```
public static Matrix dividedby( numerical.matrix.Matrix matrix,  
double scalar )
```

- **Usage**

- * Divides a matrix by a scalar and returns a matrix.

- **Parameters**

- * `matrix` - The matrix to be scaled.

- * `scalar` - The scaling factor.

-
- **Returns** - The matrix product.
-

- *dot*

```
public double dot( numerical.matrix.Matrix matrix )
```

- **Usage**

- * Computes the dot product of two matrices.

- **Parameters**

- * **matrix** - numerical.matrix.Matrix The matrix to be dotted with this one.

- **Returns** - The dot product.

- **Exceptions**

- * java.lang.Exception - If the matrices aren't the same size.
-

- *equals*

```
public void equals( numerical.matrix.Matrix matrix )
```

- **Usage**

- * Copies another matrix to this one.

- **Parameters**

- * **matrix** - The matrix to be copied into this one.
-

- *getCol*

```
public Vector getCol( int colnum )
```

- **Usage**

- * Returns the specified column of the matrix.

- **Parameters**

- * **colnum** - The column to return.

- **Returns** - The column.

- **Exceptions**

- * java.lang.Exception - If the index exceeds matrix dimensions.
-

- *getName*

```
public String getName( )
```

- **Usage**

* Returns the name of this matrix.

- **Returns** - The name of the matrix (if any).

• *getRow*

```
public Vector getRow( int rownum )
```

- **Usage**

* Returns the specified row of the matrix.

- **Parameters**

* rownum - The row to return.

- **Returns** - The row.

- **Exceptions**

* java.lang.Exception - If the index exceeds matrix dimensions.

• *incValue*

```
public void incValue( int row, int col, double value )
```

- **Usage**

* Increments the value of the matrix entry by the given amount.

- **Parameters**

* row - The row index of the value to be incremented.

* col - The column index of the value to be incremented.

* value - The value to increment the entry by.

- **Exceptions**

* java.lang.Exception - If the specified index is invalid.

• *invert*

```
public Matrix invert( )
```

- **Usage**

* Inverts a matrix.

- **Returns** - The inverse of the matrix.

- **Exceptions**

* java.lang.Exception - If the matrix isn't square.

- *isEqualTo*

```
public boolean isEqualTo( numerical.matrix.Matrix matrix )
```

- **Usage**

- * Compares two matrices.

- **Parameters**

- * The - matrix to be compared.

- **Returns** - 'True' if the matrices are equal; 'false' otherwise.

- *load*

```
public static Matrix load( java.io.FileInputStream in )
```

- **Usage**

- * Loads a matrix from a binary file.

- **Parameters**

- * out - The handle to the output file.

- **Exceptions**

- * `java.lang.Exception` - If there are imaginary values in the matrix.

- * `java.io.Exception` - If there's a problem reading from the file.

- *luBackSubstitute*

```
public Matrix luBackSubstitute( numerical.matrix.Matrix rhs )
```

- **Usage**

- * Does the backsubstitution on a matrix which has been LU-decomposed.

- **Parameters**

- * rhs - The right hand side vector.

- **Returns** - The solution.

- **Exceptions**

- * `java.lang.Exception` - If the length of the RHS doesn't match the matrix size.

- * `java.lang.Exception` - If the LU decomposition fails.

- *luDecompose*

public void **luDecompose**()

- **Usage**

- * Performs LU-decomposition on the matrix using Crout's algorithm.

- **Exceptions**

- * `java.lang.Exception` - If the matrix is singular.

- * `java.lang.Exception` - If the matrix isn't square.

- *max*

public double **max**()

- **Usage**

- * Returns the maximum value in the matrix.

- **Returns** - The maximum value in the matrix

- *maxSV*

public double **maxSV**()

- **Usage**

- * Returns the maximum singular value.

- **Returns** - The maximum singular value.

- **Exceptions**

- * `java.lang.Exception` - If the SVD fails.

- *mean*

public double **mean**()

- **Usage**

- * Returns the mean value of the matrix.

- **Returns** - The mean value in the matrix.

- *min*

public double **min**()

- **Usage**

* Returns the minimum value in the matrix.

– **Returns** - The minimum value in the matrix.

- *minSV*

public double **minSV**()

– **Usage**

* Returns the minimum singular value.

– **Returns** - The minimum singular value.

– **Exceptions**

* `java.lang.Exception` - If the SVD fails.

- *minus*

public Matrix **minus**(numerical.matrix.Matrix **matrix**)

– **Usage**

* Subtracts two matrices and returns a matrix.

– **Parameters**

* **matrix** - The matrix to be subtracted from this one.

– **Returns** - The difference of the two matrices.

– **Exceptions**

* `java.lang.Exception` - If the matrices are not the same size.

- *norm*

public double **norm**()

– **Usage**

* Same as `max()`, but with a different name. Technically only supposed to be used on column vectors.

– **Returns** - The maximum value in matrix.

- *numCols*

public int **numCols**()

– **Usage**

* Returns the number of columns in the matrix.

– **Returns** - The number of columns in the matrix.

- *numRows*

```
public int numRows( )
```

– **Usage**

* Returns the number of rows in the matrix.

– **Returns** - The number of rows in the matrix.

- *pad*

```
public void pad( int rows, int cols )
```

– **Usage**

* "Pads" the matrix to the specified size. All new entries are set to 0.0

– **Parameters**

* **rows** - The new number of rows.

* **cols** - The new number of columns.

- *pad*

```
public void pad( int rows, int cols, double value )
```

– **Usage**

* "Pads" the matrix to the specified size. New entries are set to the given value.

– **Parameters**

* **rows** - The new number of rows.

* **cols** - The new number of columns.

* **value** - The value to initialize new entries with.

- *plus*

```
public Matrix plus( numerical.matrix.Matrix matrix )
```

– **Usage**

* Adds two matrices together and returns a matrix.

– **Parameters**

- * The - matrix to be added to this one.
 - **Returns** - The sum of the two matrices
 - **Exceptions**
 - * `java.lang.Exception` - If the matrices are not the same size.
-

- *prettyPrint*

```
public void prettyPrint( )
```

- **Usage**
 - * Sends the matrix to standard output in a legible format.
-

- *rank*

```
public int rank( )
```

- **Usage**
 - * Returns the rank of the matrix.
 - **Returns** - The rank of the matrix.
 - **Exceptions**
 - * `java.lang.Exception` - If the SV decomposition fails.
-

- *returnLower*

```
public Matrix returnLower( )
```

- **Usage**
 - * Returns the lower triangular matrix.
 - **Returns** - The lower triangular matrix.
 - **Exceptions**
 - * `java.lang.Exception` - If the LU-Decomposition fails.
-

- *returnU*

```
public Matrix returnU( )
```

- **Usage**
 - * Returns the SVD "U" matrix.
- **Returns** - The "U" matrix.

- **Exceptions**

- * `java.lang.Exception` - If the SV-Decomposition fails.

- *returnUpper*

- `public Matrix returnUpper( )`

- **Usage**

- * Returns the upper triangular matrix.

- **Returns** - Upper triangular matrix.

- **Exceptions**

- * `java.lang.Exception` - If the LU-Decomposition fails.

- *returnV*

- `public Matrix returnV( )`

- **Usage**

- * Returns the SVD "V" matrix.

- **Returns** - The "V" matrix.

- **Exceptions**

- * `java.lang.Exception` - If the SV-Decomposition fails.

- *returnW*

- `public Matrix returnW( )`

- **Usage**

- * Returns the singular values.

- **Returns** - The diagonal matrix of singular values.

- **Exceptions**

- * `java.lang.Exception` - If the SV-Decomposition fails.

- *morm*

- `public double norm( )`

- **Usage**

- * Returns the max row norm.

- **Returns** - The maximum of the row sum.

- *rowSum*

public Matrix **rowSum**()

- **Usage**

- * Sums the entries in each row.

- **Returns** - Matrix of row sums.

- *save*

public void **save**(java.io.FileOutputStream **out**)

- **Usage**

- * Saves the matrix to a binary file.

- **Parameters**

- * out - The handle to the output file.

- **Exceptions**

- * java.io.Exception - If there's a problem writing to the file.

- *scale*

public void **scale**(double **scalar**)

- **Usage**

- * Scales the matrix.

- **Parameters**

- * scalar - The scaling factor.

- *scaleValue*

public void **scaleValue**(int **row**, int **col**, double **scale**)

- **Usage**

- * Scales the value of the matrix entry by the given amount.

- **Parameters**

- * row - The row index of the value to be incremented.

- * col - The column index of the value to be incremented.

- * scale - The value to scale the entry by.

- Exceptions

- * `java.lang.Exception` - If the specified index is invalid.
-

- *setCol*

```
public void setCol( int col, numerical.matrix.Vector vector )
```

- Usage

- * Sets a column of the matrix.

- Parameters

- * `col` - The index of the column to be replaced.
- * `vector` - The vector to assign to the column.

- Exceptions

- * `java.lang.Exception` - If the length of the vector doesn't match the number of rows in the matrix.
 - * `java.lang.Exception` - If the column index is invalid.
-

- *setName*

```
public void setName( java.lang.String string )
```

- Usage

- * Names the matrix.

- Parameters

- * `string` - What to name the matrix.
-

- *setRow*

```
public void setRow( int row, numerical.matrix.Vector vector )
```

- Usage

- * Sets a row of the matrix.

- Parameters

- * `row` - The index of the row to be replaced.
- * `vector` - The vector to assign to the row.

- Exceptions

- * `java.lang.Exception` - If the length of the vector doesn't match the number of columns in the matrix.

* `java.lang.Exception` - If the row index is invalid.

- *setValue*

```
public void setValue( int row, int col, double value )
```

- **Usage**

- * Sets the value of a matrix entry.

- **Parameters**

- * `row` - The row index of the value to be assigned.

- * `col` - The column index of the value to be assigned.

- * `value` - The value to be assigned.

- **Exceptions**

- * `java.lang.Exception` - If the specified index is invalid.

- *solve*

```
public Matrix solve( numerical.matrix.Matrix lhs,  
numerical.matrix.Matrix rhs )
```

- **Usage**

- * Solves a linear system of equations $A \cdot X = B$

- **Parameters**

- * `lhs` - The left hand side matrix.

- * `rhs` - The right hand side vector.

- **Returns** - The solution to the system.

- **Exceptions**

- * `java.lang.Exception` - If the LU-backsubstitution fails.

- * `java.lang.Exception` - If the LU decomposition fails.

- *submatrix*

```
public Matrix submatrix( int rowstart, int colstart, int height, int  
width )
```

- **Usage**

- * Returns the specified submatrix.

- **Parameters**

- * `rowstart` - The uppermost row of the submatrix.
- * `colstart` - The leftmost column of the submatrix.
- * `height` - The height of the submatrix.
- * `width` - The width of the submatrix.

– **Returns** - Specified submatrix.

– **Exceptions**

- * `java.lang.Exception` - If the submatrix exceeds matrix dimensions.

- *svBackSubstitute*

```
public Matrix svBackSubstitute( numerical.matrix.Matrix rhs )
```

– **Usage**

- * Do the backsubstitution on a matrix which has been SV-decomposed.

– **Parameters**

- * `rhs` - The right hand side vector.

– **Returns** - The solution.

– **Exceptions**

- * `java.lang.Exception` - The length of the RHS doesn't match the matrix size.
- * `java.lang.Exception` - If the SV decomposition fails.

- *svDecompose*

```
public void svDecompose( )
```

– **Usage**

- * Performs a singular value decomposition on the matrix.

– **Exceptions**

- * `java.lang.Exception` - If convergence fails.

- *times*

```
public static Matrix times( double scalar, numerical.matrix.Matrix  
matrix )
```

– **Usage**

- * Multiplies a matrix by a scalar and returns a matrix.

– **Parameters**

- * scalar - The scaling factor.
 - * matrix - The matrix to be scaled.
 - **Returns** - The matrix product.
-

- *times*

```
public Matrix times( numerical.matrix.Matrix matrix )
```

- **Usage**

- * Multiplies two matrices and returns a matrix.

- **Parameters**

- * The - matrix to be multiplied by this one.

- **Returns** - The product of the two matrices.

- **Exceptions**

- * `java.lang.Exception` - If the inner dimensions of the two matrices don't match.
-

- *times*

```
public static Matrix times( numerical.matrix.Matrix matrix, double  
scalar )
```

- **Usage**

- * Multiplies a matrix by a scalar and returns a matrix.

- **Parameters**

- * `matrix` - The matrix to be scaled.
 - * `scalar` - The scaling factor.

- **Returns** - The matrix product.

- *times*

```
public Matrix times( numerical.matrix.Vector vector )
```

- **Usage**

- * Multiplies a matrix by a vector and returns a matrix.

- **Parameters**

- * `vector` - Vector to be multiplied by this matrix.

- **Returns** - Matrix product of the vector and the matrix.

- **Exceptions**

- * `java.lang.Exception` - If the inner dimensions of the matrix and the vector don't match.
-

- *toDoubleArray*

```
public double toDoubleArray( )
```

- **Usage**

- * Returns the matrix as an array of doubles.

- **Returns** - The array of doubles.

- *toString*

```
public String toString( )
```

- **Usage**

- * Returns the matrix as a formatted string.

- **Returns** - Matrix entries, nicely formatted.

- *trace*

```
public double trace( )
```

- **Usage**

- * Returns the trace of the matrix.

- **Returns** - The trace of the matrix. (The sum of the diagonal elements.)

- **Exceptions**

- * `java.lang.Exception` - If the matrix is not square.
-

- *transpose*

```
public void transpose( )
```

- **Usage**

- * Transposes the matrix in place.
-

- *transpose*

```
public static Matrix transpose( numerical.matrix.Matrix matrix )
```

– **Usage**

* Returns the transpose of the matrix.

– **Parameters**

* *matrix* - The matrix to transpose.

– **Returns** - The transpose of the input.

• *trim*

```
public void trim( int rows, int cols )
```

– **Usage**

* "Trims" the matrix to the specified size.

– **Parameters**

* *rows* - The new number of rows.

* *cols* - The new number of columns.

– **Exceptions**

* `java.lang.Exception` - If the size exceeds the matrix dimensions.

• *value*

```
public double value( int row, int col )
```

– **Usage**

* Returns the value of the matrix entry.

– **Parameters**

* *row* - The row index of the value to be returned.

* *col* - The column index of the value to be returned.

– **Returns** - The value of the specified matrix entry.

– **Exceptions**

* `java.lang.Exception` - If the specified index is invalid.

G.1.2 Class **Vector**

This is a vector class.

Declaration

```
public class Vector
extends numerical.matrix.Matrix
```

Constructors

• *Vector*

```
public Vector( double [] value )
```

– **Usage**

* Creates a vector from the array of doubles.

– **Parameters**

* **value** - The array of doubles to fill the vector with.

• *Vector*

```
public Vector( int rows )
```

– **Usage**

* Creates a vector of the specified length and initializes it to 0.0.

– **Parameters**

* **rows** - The length of the vector.

• *Vector*

```
public Vector( numerical.matrix.Matrix matrix )
```

– **Usage**

* Creates a vector from the first column or first row, whichever is longer.

– **Parameters**

* **matrix** - The matrix to use.

• *Vector*

```
public Vector( numerical.matrix.Vector vector )
```

– **Usage**

* Creates a new vector from the given one.

– **Parameters**

- * `vector` - The vector to copy.

Methods

- *incValue*

public void **incValue**(int **row**, double **value**)

– **Usage**

- * Increments the vector entry by the specified value.

– **Parameters**

- * `row` - The entry to change.
- * `value` - The value to increment it by.

– **Exceptions**

- * `java.lang.Exception` - If the specified index exceeds the vector length.
-

- *length*

public int **length**()

– **Usage**

- * Returns the length of the vector.

– **Returns** - The length of the vector.

- *inspace*

public static Vector **inspace**(double **start**, double **end**, int **numsteps**)

– **Usage**

- * Creates a linearly spaced vector.

– **Parameters**

- * `start` - The value of the first entry.
- * `end` - The value of the last entry.
- * `numsteps` - The number of elements in the vector.

– **Returns** - A linearly spaced vector, from start to end.

- *setValue*

```
public void setValue( int row, double value )
```

- **Usage**

- * Sets the vector entry to the specified value.

- **Parameters**

- * **row** - The entry to change.

- * **value** - The new value.

- **Exceptions**

- * `java.lang.Exception` - If the specified index exceeds the vector length.

- *times*

```
public Matrix times( numerical.matrix.Matrix matrix )
```

- **Usage**

- * Multiplies the vector by a matrix.

- **Parameters**

- * **matrix** - The matrix to multiply by the vector.

- **Returns** - The resultant matrix.

- **Exceptions**

- * `java.lang.Exception` - If the vector length doesn't match the number of rows in the matrix.

- *toString*

```
public String toString( )
```

- **Usage**

- * Returns the vector as a formatted string.

- **Returns** - Vector entries, nicely formatted.

- *value*

```
public double value( int row )
```

- **Usage**

- * Returns the value of the specified vector entry.

– Parameters

- * `row` - The index of the value to return.

– Exceptions

- * `java.lang.Exception` - If the specified index exceeds the vector length.

Appendix H

Package numerical.workspace

<i>Package Contents</i>	<i>Page</i>
Classes	
Workspace	217

H.1 Classes

H.1.1 Class **Workspace**

This is a linked list of matrices, vectors, or neural networks.

Declaration

```
public class Workspace  
extends mjc.util.Group
```

Constructors

- *Workspace*
public **Workspace**()
 - **Usage**
 - * Creates a new, empty workspace.

Methods

- *add*
public void **add**(numerical.matrix.Matrix **matrix**)
 - **Usage**
 - * Adds a matrix to the workspace.
 - **Parameters**
 - * **matrix** - The matrix to be added.
-
- *add*
public void **add**(ai.neural.network.Network **network**)
 - **Usage**
 - * Adds a neural network to the workspace.
 - **Parameters**
 - * **network** - The network to be added.

- *add*

```
public void add( java.lang.Object object )
```

- **Usage**

- * Traps an attempt to add a generic object to the workspace.

- **Parameters**

- * *object* - The object to be added.

- **Exceptions**

- * *java.lang.Exception* - Always. Arbitrary objects cannot be added to the workspace.

- *load*

```
public static Workspace load( java.io.FileInputStream in )
```

- **Usage**

- * Loads a workspace from a binary file.

- **Parameters**

- * *in* - The handle to the input file.

- *loadMatLabWorkspace*

```
public static Workspace loadMatLabWorkspace( java.io.FileInputStream  
in )
```

- **Usage**

- * Loads a Matlab 4.0 workspace.

- **Parameters**

- * *in* - The handle to the input file.

- *save*

```
public void save( java.io.FileOutputStream out )
```

- **Usage**

- * Saves a workspace to a binary file.

- **Parameters**

* out - The handle to the output file.

- *saveMatLabWorkspace*

```
public void saveMatLabWorkspace( java.io.FileOutputStream out )
```

- **Usage**

- * Save the workspace as a Matlab 4.0 workspace. Only matrices and vectors are saved.

- **Parameters**

- * out - The handle to the output file.

Appendix I

Package numerical.integration

<i>Package Contents</i>	<i>Page</i>
<hr/>	
Interfaces	
ODEFunction	221
Classes	
IntegratorClient	222
IntegratorServer	224
Integrator	226
<hr/>	

I.1 Interfaces

I.1.1 Interface **ODEFunction**

This is the interface any state function has to provide in order to be used by the integration routine.

Declaration

```
public interface ODEFunction
```

Methods

- *compute*

```
public Vector compute( )
```

- **Usage**

- * Returns the initial conditions.

- **Returns** - The initial conditions.

- *compute*

```
public Vector compute( double t, numerical.matrix.Vector y )
```

- **Usage**

- * Computes the state derivatives.

- **Parameters**

- * t - Current simulation time.

- * y - Current states.

- **Returns** - The state derivatives.

- *computeJacobian*

```
public Matrix computeJacobian( double t, double h,  
numerical.matrix.Vector y )
```

- **Usage**

- * Computes the Jacobian matrix.

- **Parameters**

- * *t* - Current simulation time.
 - * *h* - Current step size.
 - * *y* - Current states
 - **Returns** - Matrix with the first column representing df/dt and the remaining square matrix representing df/dy .
-

- *numStates*

```
public int numStates( )
```

- **Usage**
 - * Returns the number of states.
 - **Returns** - The number of states.
-

- *toString*

```
public String toString( )
```

- **Usage**
 - * Returns a string representation of the states
- **Returns** - A string representation of the states.

I.2 Classes

I.2.1 Class **IntegratorClient**

This class does the integration for one multi-step (of pre-determined step size.)

Declaration

```
public class IntegratorClient
extends java.lang.Thread
```

Constructors

- *IntegratorClient*

```
public IntegratorClient( numerical.integration.ODEFunction odef )
```

- **Usage**

* Creates a integrator client for the given function.

– **Parameters**

* odef - The function to be integrated.

• *IntegratorClient*

```
public IntegratorClient( numerical.integration.ODEFunction odef,  
boolean usesvd )
```

– **Usage**

* Creates an integrator client for the given function..

– **Parameters**

* odef - The function to be integrated.

* usesvd - 'True' uses SV instead of LU decomposition; 'false' does the opposite.

Methods

• *returnDebugInfo*

```
public Vector returnDebugInfo( )
```

– **Usage**

* Returns the debug info for the integration.

– **Returns** - The debug information.

• *setDebugMode*

```
public void setDebugMode( boolean yesno )
```

– **Usage**

* Turns integration debug mode on or off. Debug mode will generate an additional output matrix which can be used to resolve integration problems.

– **Parameters**

* yesno - 'True' turns on debug mode; 'false' turns it off.

- *simpr*

```
public Vector simpr( numerical.matrix.Vector y,  
numerical.matrix.Vector dy_dt, numerical.matrix.Vector df_dt,  
numerical.matrix.Matrix df_dy, double tStart, double hTotal, int  
numSteps )
```

- **Usage**

- * Takes an integration step.

- **Parameters**

- * y - The current states.
 - * dy_dt - The state derivatives.
 - * df_dt - The derivatives of the RHS.
 - * df_dy - The Jacobian.
 - * tStart - The start time for the step.
 - * hTotal - The total step to be taken.
 - * numSteps - The number of substeps to be used.

- **Returns** - The new states.

- **Exceptions**

- * java.lang.Exception - If something goes wrong internally.

1.2.2 Class **IntegratorServer**

This class coordinates the integration between the various clients.

Declaration

```
public class IntegratorServer  
extends java.lang.Thread
```

Constructors

- *IntegratorServer*

```
public IntegratorServer( numerical.integration.ODEFunction odef )
```

- **Usage**

- * Creates a new integrator server.

- Parameters

- * `odef` - The function to be integrated.

Methods

- *doStep*

```
public void doStep( double t, numerical.matrix.Vector y,  
numerical.matrix.Vector dy_dt, double h, double errortolerance,  
numerical.matrix.Vector yscal )
```

- Usage

- * Takes a single step.

- Parameters

- * `t` - The current simulation time.
- * `y` - The current states.
- * `dy_dt` - The state derivatives.
- * `h` - The step size.
- * `errortolerance` - The error tolerance.
- * `yscal` - The scaling vector for the states. (To scale the integration error.)

- Exceptions

- * `java.lang.Exception` - If something goes wrong internally.
-

- *returnDebugInfo*

```
public Vector returnDebugInfo( )
```

- Usage

- * Returns the debug info for the integration.

- Returns - The debug information.

- *returnNextStepSize*

```
public double returnNextStepSize( )
```

- Usage

- * Returns the next suggested step size for the integration.

- Returns - The next step size.

- *returnStepSize*

```
public double returnStepSize( )
```

- **Usage**

- * Returns the step size taken the preceeding step.

- **Returns** - The step size taken.

- *returnYVector*

```
public Vector returnYVector( )
```

- **Usage**

- * Returns the state vector for the integration step.

- **Returns** - The new states.

- *setDebugMode*

```
public void setDebugMode( boolean yesno )
```

- **Usage**

- * Turns integration debug mode on or off. Debug mode will generate an additional output matrix which can be used to resolve integration problems.

- **Parameters**

- * **yesno** - 'True' turns on debug mode; 'false' turns it off.

I.2.3 Class **Integrator**

This is the "driver" for the integration.

Declaration

```
public class Integrator  
extends java.lang.Object
```

Constructors

- *Integrator*

```
public Integrator( numerical.integration.ODEFunction odef )
```

- **Usage**

- * Creates an integrator object. Note that start time, end time, and initial conditions must be set before integration can be performed.

- **Parameters**

- * **odef** - The function to be integrated.
-

- *Integrator*

```
public Integrator( numerical.integration.ODEFunction odef, double  
start, double end, numerical.matrix.Vector ics )
```

- **Usage**

- * Creates an integrator object.

- **Parameters**

- * **odef** - The function to be integrated.
 - * **start** - The integration start time.
 - * **end** - The integration end time.
 - * **ics** - The initial conditions.
-

- *Integrator*

```
public Integrator( numerical.matrix.Vector ics )
```

- **Usage**

- * Creates and integrator object. Note that ODEFunction, start time and end time must be set before integration can be performed.

- **Parameters**

- * **ics** - The initial conditions.

Methods

- *returnDebugInfo*

```
public Matrix returnDebugInfo( )
```

- **Usage**

- * Returns the matrix of debug information.

- **Returns** - The matrix of debug information for the integration.

- *returnTVector*

```
public Vector returnTVector( )
```

- **Usage**

- * Returns the vector of integration timesteps.

- **Returns** - The integration times.

- *returnYMatrix*

```
public Matrix returnYMatrix( )
```

- **Usage**

- * Returns a matrix of the states' values at the various integration timesteps.

- **Returns** - Matrix of states for the integration times.

- *setConsoleDisplayTimeOnly*

```
public void setConsoleDisplayTimeOnly( boolean yesno )
```

- **Usage**

- * Sets the console to display only integration time during the course of the integration. (As opposed to both the current integration time and the current states.)

- **Parameters**

- * **yesno** - 'Yes' sets the console display to time only; 'false' turns the regular display on.

- *setConsoleOutput*

```
public void setConsoleOutput( boolean yesno )
```

- **Usage**

- * Turns console output on or off.

- **Parameters**

* *yesno* - 'True' turns on console output; 'false' turns it off.

- *setDebugMode*

public void **setDebugMode**(boolean *yesno*)

- **Usage**

- * Turns integration debug mode on or off. Debug mode will generate an additional output matrix which can be used to resolve integration problems.

- **Parameters**

- * *bool* - 'True' turns on debug mode; 'false' turns it off.

- *setEndTime*

public void **setEndTime**(double *end*)

- **Usage**

- * Sets the integration end time.

- **Parameters**

- * *end* - The new end time.

- *setErrorTolerance*

public void **setErrorTolerance**(double *error*)

- **Usage**

- * Sets the error tolerance.

- **Parameters**

- * *error* - The new error tolerance.

- *setInitialConditions*

public void **setInitialConditions**(numerical.matrix.Vector *ics*)

- **Usage**

- * Sets the initial conditions.

- **Parameters**

- * *ics* - The new initial conditions.

- *setInitialStepSize*

```
public void setInitialStepSize( double iss )
```

- **Usage**

- * Sets the initial step size for the integration.

- **Parameters**

- * **iss** - The new initial step size.

- *setMaximumNumberOfSteps*

```
public void setMaximumNumberOfSteps( int mns )
```

- **Usage**

- * Sets the maximum number of steps to try before giving up.

- **Parameters**

- * **mns** - The maximum number of steps.

- *setMaximumStepSize*

```
public void setMaximumStepSize( double mss )
```

- **Usage**

- * Sets the maximum step size.

- **Parameters**

- * **mss** - Maximum step size.

- *setMinimumStepSize*

```
public void setMinimumStepSize( double mss )
```

- **Usage**

- * Sets the minimum step size.

- **Parameters**

- * **mss** - Minimum step size.

- *setODEFunction*

```
public void setODEFunction( numerical.integration.ODEFunction odef )
```

– **Usage**

* Sets the ODEFunction to be integrated.

– **Parameters**

* odef - An ODEFunction object to be integrated.

- *setOutput*

```
public void setOutput( graphics.graph.GraphPane graphpane )
```

– **Usage**

* Sets the graph to use when plotting the integration progress.

– **Parameters**

* graphpane - The graph to plot the integration data on

- *setSaveEvery*

```
public void setSaveEvery( int increment )
```

– **Usage**

* Sets the save-to-disk increment for the integration. (Useful for recovering from system crashes.)

– **Parameters**

* increment - The number of steps to save to disk at a time.

- *setSaveEverySingleStep*

```
public void setSaveEverySingleStep( boolean yesno )
```

– **Usage**

* Determines whether or not the integration routine should record the state values for every single integration step, as opposed to discarding intermediate values..

– **Parameters**

* yesno - 'True' sets the integration routine to record the state values at every step. 'False' allows the integration routine to discard intermediate values.

- *setStartTime*

```
public void setStartTime( double start )
```

– **Usage**

- * Sets the integration start time.

– **Parameters**

- * `start` - Start time.
-

- *setStorageIncrement*

public void **setStorageIncrement**(int **increment**)

– **Usage**

- * Determines how much the storage matrices are padded by each time they fill.

– **Parameters**

- * `increment` - How much to pad the matrices by when we run out of space
-

- *setTIncrementToSave*

public void **setTIncrementToSave**(double **increment**)

– **Usage**

- * Sets the save increment for integration time. (i.e., the independent variable.)

– **Parameters**

- * `increment` - The save increment.
-

- *start*

public void **start**()

– **Usage**

- * Starts the integration.

– **Exceptions**

- * `java.lang.Exception` - If the function has not been specified.
- * `java.lang.Exception` - If the start time is greater than the end time.
- * `java.lang.Exception` - If the initial conditions have not been specified.
- * `java.lang.Exception` - If the initial conditions do not match the function.

Appendix J

Package numerical.formatting

<i>Package Contents</i>	<i>Page</i>
Classes	
NumberFormat	234

J.1 Classes

J.1.1 Class **NumberFormat**

Formats a number to make the console output pretty.

Declaration

```
public class NumberFormat  
extends java.lang.Object
```

Constructors

- *NumberFormat*
 public **NumberFormat**()

Methods

- *formatDouble*
 public static String **formatDouble**(double **x**)
 - **Usage**
 - * Format a double into a string.
 - **Parameters**
 - * **x** - The number to be formatted.
 - **Returns** - The string.
- *formatFixed*
 public static String **formatFixed**(double **d**)
 - **Usage**
 - * Format the fixed portion of the number.
 - **Parameters**
 - * **d** - The number to be formatted.
 - **Returns** - The string.

Appendix K

Package numerical.machine

<i>Package Contents</i>	<i>Page</i>
Classes	
Machine	236

K.1 Classes

K.1.1 Class **Machine**

This class computes (and stores) the machine-specific parameters affecting floating point arithmetic.

Declaration

```
public class Machine  
extends java.lang.Object
```

Constructors

- *Machine*

```
public Machine( )
```

- **Usage**

- * Computes and stores the machine specific paramters.

Methods

- *getEpsilon*

```
public static double getEpsilon( )
```

- **Usage**

- * Returns the machine epsilon.

- **Returns** - The machine epsilon.

- *reinitialize*

```
public void reinitialize( )
```

- **Usage**

- * Computes the machine parameters again.

Appendix L

Package mjc.util

<i>Package Contents</i>	<i>Page</i>
<hr/>	
Interfaces	
BidirectionalEnumeration	238
Comparator	239
Classes	
Group	240
GenericComparator	242
Group.Link	243
<hr/>	

L.1 Interfaces

L.1.1 Interface **BidirectionalEnumeration**

The bidirectional enumeration generates a series of elements which can be traversed in both directions.

Declaration

```
public interface BidirectionalEnumeration  
implements java.util.Enumeration
```

Methods

- *gotoHead*

```
public void gotoHead( )
```

- **Usage**

- * Jumps to the first element in the enumeration.

- *gotoTail*

```
public void gotoTail( )
```

- **Usage**

- * Jumps to the last element in the enumeration.

- *hasMorePrevElements*

```
public boolean hasMorePrevElements( )
```

- **Usage**

- * Tests if this enumeration has more "previous" elements.

- **Returns** - 'True' if this enumeration contains more elements; 'false' otherwise.

- *prevElement*

```
public Object prevElement( )
```

- **Usage**

- * Returns the previous element of this enumeration.

- **Returns** - The next element of this enumeration.
- **Exceptions**
 - * `java.util.NoSuchElementException` - If no more elements exist.

L.1.2 Interface **Comparator**

A comparator allows a comparison between two like objects.

Declaration

`public interface Comparator`

Methods

- *equalTo*

```
public boolean equalTo( java.lang.Object a, java.lang.Object b )
```

- **Usage**

- * Checks to see if two objects are equal.

- **Parameters**

- * **a** - The first object.
 - * **b** - The second object.

- **Returns** - 'True' if 'a' and 'b' are references to the same object; 'false' otherwise.
-

- *greaterThan*

```
public boolean greaterThan( java.lang.Object a, java.lang.Object b )
```

- **Usage**

- * Compares the order of two objects.

- **Parameters**

- * **a** - The first object.
 - * **b** - The second object.

- **Returns** - 'True' if 'a' has a higher rank than 'b'; false otherwise.
-

- *lessThan*

```
public boolean lessThan( java.lang.Object a, java.lang.Object b )
```

- **Usage**

- * Compares the order of two objects.

- **Parameters**

- * a - The first object.

- * b - The second object.

- **Returns** - 'True' if 'a' has a lower rank than 'b'; 'false' otherwise.

- *typeCheck*

```
public void typeCheck( java.lang.Object a )
```

- **Usage**

- * Performs type-checking on the input object.

- **Parameters**

- * a - The object to be tested.

- **Exceptions**

- * java.lang.RuntimeException - If the object's type is incorrect.

L.2 Classes

L.2.1 Class Group

A group is an ordered collection of like objects.

Declaration

```
public class Group  
extends java.lang.Object  
implements java.io.Serializable
```

Constructors

- *Group*

```
public Group( mjc.util.Comparator c )
```

- **Usage**

- * Creates an empty group.

- **Parameters**

- * *c* - The comparator to be used to order the group.

Methods

- *add*

```
public void add( java.lang.Object o )
```

- **Usage**

- * Adds an element to the group.

- **Parameters**

- * *o* - The element to be added.
-

- *delete*

```
public boolean delete( java.lang.Object o )
```

- **Usage**

- * Deletes and object from the group.

- **Parameters**

- * *o* - The object to delete.

- **Returns** - 'True' if the object was deleted; 'false' otherwise.

- *elements*

```
public synchronized BidirectionalEnumeration elements( )
```

- **Usage**

- * Returns a bidirectional enumeration for the group.

- **Returns** - An enumeration of the group.

- *size*

```
public int size( )
```

- **Usage**

- * Returns the number of elements in the group.
- **Returns** - The number of elements in the group.

L.2.2 Class **GenericComparator**

Allows comparison between two generic objects.

Declaration

```
public class GenericComparator
extends java.lang.Object
implements Comparator, java.io.Serializable
```

Constructors

- *GenericComparator*
public **GenericComparator**()

Methods

- *equalTo*
public boolean **equalTo**(java.lang.Object **a**, java.lang.Object **b**)
 - **Usage**
 - * This method always returns 'false'.
 - **Parameters**
 - * **a** - The first object.
 - * **b** - The second object.
 - **Returns** - 'False' for all objects.

- *greaterThan*
public boolean **greaterThan**(java.lang.Object **a**, java.lang.Object **b**)
 - **Usage**

- * This method always returns 'true'.

- **Parameters**

- * a - The first object.

- * b - The second object.

- **Returns** - 'True' for all objects.

- *lessThan*

```
public boolean lessThan( java.lang.Object a, java.lang.Object b )
```

- **Usage**

- * This method always returns 'false'.

- **Parameters**

- * The - first object.

- * The - second object.

- **Returns** - 'False' for all objects.

- *typeCheck*

```
public void typeCheck( java.lang.Object a )
```

- **Usage**

- * Performs type checking on the input object.

- **Parameters**

- * a - The object to be tested.

L.2.3 Class **Group.Link**

The link class is a doubly-linked list that keeps track of the data in a group.

Declaration

```
public class Group.Link  
extends java.lang.Object  
implements java.io.Serializable
```

Methods

- *toString*

```
public String toString( )
```

- **Usage**

- * Returns a string representing the link.

- **Returns** - A text string representing the link.

Appendix M

Package `graphics.graph`

<i>Package Contents</i>	<i>Page</i>
Classes	
GraphFrame	246
GraphPane	246

M.1 Classes

M.1.1 Class **GraphFrame**

A graph frame contains one or more graph panes.

Declaration

```
public class GraphFrame  
extends java.awt.Frame
```

Constructors

- *GraphFrame*

```
public GraphFrame( java.lang.String string )
```

- **Usage**

- * Creates a generic graph frame with the specified title.

- **Parameters**

- * *string* - The title for the graph frame.

Methods

- *handleEvent*

```
public boolean handleEvent( java.awt.Event event )
```

- **Usage**

- * This method receives all events generated within the graph frame.

- **Parameters**

- * *event* - The generated event.

- **Returns** - 'True' if the event was handled, 'false' otherwise.

M.1.2 Class **GraphPane**

The graph pane contains a graph.

Declaration

```
public class GraphPane
extends java.awt.Canvas
implements java.lang.Runnable
```

Constructors

• *GraphPane*

```
public GraphPane( )
```

– Usage

* Creates a figure with axes 0:1,0:1

• *GraphPane*

```
public GraphPane( double [] x, double [] y )
```

– Usage

* Creates a figure with the specified axes

– Parameters

* **x** - Array of two values, **xmin** and **xmax**.

* **y** - Array of two values, **ymin** and **ymax**.

• *GraphPane*

```
public GraphPane( double xmin, double xmax, double ymin, double ymax )
```

– Usage

* Creates a figure with the specified axes.

– Parameters

* **xmin** - Minimum x-value.

* **xmax** - Maximum x-value.

* **ymin** - Minimum y-value.

* **ymax** - Maximum y-value.

Methods

- *addData*

```
public synchronized void addData( double  x, double  y )
```

- **Usage**

- * Adds data points to the plot.

- **Parameters**

- * *x* - The x-coordinate of the point to add.

- * *y* - The y-coordinate of the point to add.

- **Exceptions**

- * `java.lang.Exception` - If the current plot has more than one set of 'y' data.

- *addData*

```
public synchronized void addData( double  x, numerical.matrix.Vector  
y )
```

- **Usage**

- * Adds data points to the plot

- **Parameters**

- * *x* - The x-coordinate of the data set.

- * *y* - The y-coordinates of the data set.

- **Exceptions**

- * `java.lang.Exception` - if an incorrect number of 'y' values are specified.

- *paint*

```
public void paint( java.awt.Graphics  g )
```

- **Usage**

- * Repaints the figure.

- **Parameters**

- * *The* - graphics context to paint.

- *plot*

```
public synchronized void plot( numerical.matrix.Matrix x,  
numerical.matrix.Matrix y )
```

- **Usage**

- * Plots data on the graph.

- **Parameters**

- * **x** - The x-data to plot.

- * **y** - The y-data to plot.

- **Exceptions**

- * `java.lang.Exception` - If the 'x' and 'y' data aren't the same length.

- *run*

```
public void run( )
```

- **Usage**

- * This is the graphing thread. It calls `paint()` every 20ms.

- *setRange*

```
public void setRange( double [] x, double [] y )
```

- **Usage**

- * Sets the axes' ranges.

- **Parameters**

- * **x** - The new x-range.

- * **y** - The new y-range.

- **Exceptions**

- * `java.lang.Exception` - If less than two values are specified for either 'x' or 'y'.

- *setTitle*

```
public void setTitle( java.lang.String string )
```

- **Usage**

- * Sets the title of the graph.

- **Parameters**

- * `string` - The title.

- *setXLabel*

- `public void setXLabel( java.lang.String string )`

- **Usage**

- * Sets the label for the x-axis.

- **Parameters**

- * `string` - The x-label.

- *setYLabel*

- `public void setYLabel( java.lang.String string )`

- **Usage**

- * Sets the label for the y-axis.

- **Parameters**

- * `string` - The y-label

- *start*

- `public void start( )`

- **Usage**

- * Starts the thread for this figure.

- *stop*

- `public void stop( )`

- **Usage**

- * Stops the thread for this figure.

- *update*

- `public synchronized void update( java.awt.Graphics g )`

- **Usage**

- * The update method is called whenever the image needs to be painted.

– Parameters

- * The - graphics context to paint.