

# Discrete-Time Physics-Based Modeling

A Block-Based Multi-Paradigm Approach  
with Applications to Audio and Acoustics

Matti Karjalainen

February 13, 2008



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	General concepts of physics-based modeling . . . . .	12
1.1.1	Physical domains, variables, and parameters . . . . .	12
1.1.2	Physical structure and interaction . . . . .	13
1.1.3	Signals, signal processing, and discrete-time modeling . . . . .	14
1.1.4	Linearity and time-invariance . . . . .	14
1.1.5	Energetic behaviour and stability . . . . .	15
1.1.6	Modularity and locality of computation . . . . .	16
1.1.7	Complexity in physics-based modeling . . . . .	16
1.2	Overview of paradigms for physics-based discrete-time modeling . . . . .	17
1.2.1	Modeling paradigms . . . . .	17
1.2.2	Modeling typology . . . . .	19
<b>2</b>	<b>Basics of Physics:Electrical, Mechanical, and Acoustical Systems</b>	<b>21</b>
2.1	Mathematical preliminaries . . . . .	22
2.1.1	Differential equations . . . . .	22
2.1.2	Integral transforms . . . . .	23
2.2	Electric systems . . . . .	25
2.2.1	Concepts and quantities . . . . .	25
2.2.2	Passive lumped circuit elements . . . . .	26
2.2.3	Basic circuit and network theory . . . . .	30
2.2.4	Distributed electrical systems / delay lines . . . . .	34
2.2.5	Nonlinear elements . . . . .	34
2.2.6	Active circuits . . . . .	34
2.2.7	Other examples ??? . . . . .	34
2.2.8	Circuit simulators . . . . .	34
2.3	Mechanical systems . . . . .	34
2.3.1	Mechanical quantities . . . . .	35
2.3.2	Mechanical elements . . . . .	35
2.3.3	A simple vibrating system . . . . .	35
2.3.4	Mechanical resonance . . . . .	37
2.3.5	From vibration to waves ????. . . . .	37
2.3.6	Complex mass-spring systems . . . . .	38
2.3.7	Modal behavior . . . . .	38
2.3.8	Examples ?? . . . . .	40
2.4	Acoustical systems . . . . .	40
2.4.1	From vibration to wave behavior of sound . . . . .	40

2.4.2	Waves and wave propagation . . . . .	40
2.4.3	Reflection and scattering . . . . .	40
2.4.4	Sound sources and radiation . . . . .	40
2.4.5	Acoustic elements and systems . . . . .	40
2.4.6	Acoustical resonance . . . . .	41
2.4.7	Examples ??? . . . . .	41
2.5	Analogies and equivalent circuits . . . . .	41
2.6	Transducers: coupling between physical domains . . . . .	42
2.6.1	Electromagnetic-mechanical transduction . . . . .	42
2.6.2	Mechano-acoustical transduction . . . . .	43
2.6.3	Dynamic loudspeaker . . . . .	43
2.6.4	Enclosures . . . . .	45
2.6.5	Headphones . . . . .	46
2.6.6	Electrostatic-mechanical transducer . . . . .	47
2.6.7	Other transducer types / WHERE THIS ??? . . . . .	47
2.6.8	Microphones . . . . .	47
2.7	Room acoustics . . . . .	48
2.8	Physics of musical instruments ??? . . . . .	48
2.8.1	Wind instruments ??? . . . . .	50
2.8.2	Other musical instruments ??? . . . . .	50
2.9	Room acoustics . . . . .	50
2.10	Acoustics of speech production . . . . .	50
2.10.1	Speech production mechanism . . . . .	50
2.10.2	Modeling of speech production . . . . .	52
2.11	Summary . . . . .	56
<b>3</b>	<b>Basics of Signal Processing</b>	<b>57</b>
3.1	Signals . . . . .	57
3.1.1	Important signal types . . . . .	59
3.2	Fundamental concepts of signal processing . . . . .	59
3.2.1	Linear and time-invariant systems . . . . .	59
3.2.2	Convolution . . . . .	60
3.2.3	Signal transforms . . . . .	61
3.2.4	Fourier analysis and synthesis . . . . .	61
3.2.5	Spectrum analysis . . . . .	63
3.2.6	Time-frequency representations . . . . .	64
3.2.7	Wavelets and filterbanks . . . . .	66
3.2.8	Auto- and cross-correlation . . . . .	66
3.2.9	Cepstrum . . . . .	68
3.3	Digital signal processing (DSP) . . . . .	68
3.3.1	Sampling and signal conversion . . . . .	68
3.3.2	Z-transform . . . . .	69
3.3.3	Filters as LTI systems . . . . .	70
3.3.4	Digital filtering . . . . .	70
3.3.5	Linear prediction . . . . .	71
3.4	Statistical signal processing . . . . .	73
3.5	Measurement of system response . . . . .	73

3.6	Adaptive, learning, and nonlinear systems . . . . .	73
3.6.1	Nonlinear filtering . . . . .	73
3.7	Continuous versus discrete-time models . . . . .	73
3.7.1	Backward difference approximation of derivatives . . . . .	74
3.7.2	Impulse invariant transform . . . . .	74
3.7.3	Bilinear transform . . . . .	74
3.8	Frequency warping . . . . .	75
<b>4</b>	<b><i>Distributed W-Modeling:Digital Waveguide (DWG) Models</i></b>	<b>77</b>
4.1	DWG theory . . . . .	77
4.1.1	Discretization of wave propagation . . . . .	77
4.1.2	Modeling of losses and dispersion . . . . .	79
4.1.3	Simple DWG string model . . . . .	80
4.1.4	Parallel DWG junction . . . . .	81
4.1.5	Series DWG junction . . . . .	82
4.1.6	DWG connectivity . . . . .	83
4.2	Some practical DWG structures . . . . .	83
4.2.1	Scattering at impedance discontinuity: Kelly-Lochbaum junction . . . . .	84
4.2.2	Interpolated KL-junction . . . . .	85
4.2.3	Interpolated side branch . . . . .	85
4.3	Digital waveguide meshes and networks . . . . .	86
4.3.1	Discussion and summary . . . . .	86
<b>5</b>	<b><i>Distributed K-modeling:Finite Difference Time Domain (FDTD) Models</i></b>	<b>89</b>
5.0.2	Source terms . . . . .	90
5.0.3	Admittance discontinuity and scattering . . . . .	91
5.0.4	Continuity laws . . . . .	92
<b>6</b>	<b><i>Lumped W-Modeling:Wave Digital Filter (WDF) Models</i></b>	<b>95</b>
6.1	WDF theory . . . . .	95
6.1.1	Physically normalized waves . . . . .	96
6.1.2	Power-normalized waves . . . . .	97
6.1.3	Connectivity of wave ports . . . . .	97
6.2	Basic WDF elements . . . . .	98
6.2.1	Resistance . . . . .	99
6.2.2	Conductance . . . . .	99
6.2.3	Voltage source . . . . .	100
6.2.4	Current source . . . . .	100
6.2.5	Capacitance . . . . .	101
6.2.6	Inductance . . . . .	104
6.2.7	Generalized first order reflectance . . . . .	104
6.3	WDF elements with I-port . . . . .	105
6.3.1	Resistance with I-port . . . . .	105
6.3.2	Voltage and current sources with I-port . . . . .	106
6.3.3	Example of simple nonlinearity: ideal diode . . . . .	106
6.4	Adaptors . . . . .	107
6.4.1	Parallel N-port adaptor . . . . .	107

6.4.2	Series N-port adaptor . . . . .	109
6.4.3	3-port adaptors . . . . .	111
6.4.4	Parallel 2-port adaptor . . . . .	111
6.4.5	Series 2-port adaptor . . . . .	112
6.4.6	WDF adaptors vs. DWG scattering junctions . . . . .	112
6.5	Network structures and interconnection rules . . . . .	113
6.5.1	Tree structures . . . . .	113
6.5.2	Binary trees . . . . .	114
6.5.3	Root elements of a WDF tree structure . . . . .	114
6.5.4	Initialization of WDF networks . . . . .	114
6.6	WDF 2-port and N-port elements . . . . .	115
6.6.1	Ideal transformer . . . . .	116
6.6.2	Gyrator . . . . .	117
6.6.3	Dualizer . . . . .	117
6.6.4	Circulator . . . . .	117
6.6.5	Unit element . . . . .	118
6.6.6	Mutators and resistance-reactance transforms . . . . .	118
6.7	Transducers as inter-domain two-port elements . . . . .	120
6.7.1	Example: Electrodynamic transduction . . . . .	121
6.8	K-W conversion in lumped element models . . . . .	122
6.8.1	Impedance given in polynomial form . . . . .	123
6.8.2	Impedance given in rational form . . . . .	124
6.8.3	Second order sections . . . . .	124
6.8.4	Lattice formulations . . . . .	124
6.8.5	Impedance given in parallel form . . . . .	124
6.8.6	Impedance given in cascaded form ??? . . . . .	126
6.9	Conversion of state-space forms to W-elements . . . . .	126
<b>7</b>	<b>KW-Compatibility and Hybrid Modeling</b>	<b>127</b>
7.1	KW conversion between DWGs and FDTDs . . . . .	127
7.1.1	Construction of hybrid models ??? . . . . .	127
<b>8</b>	<b>Lumped K-Modeling: Various Techniques</b>	<b>129</b>
8.1	Modal decomposition techniques . . . . .	129
8.2	Mass-spring models . . . . .	129
<b>9</b>	<b>Time-Variant and Nonlinear Modeling</b>	<b>131</b>
9.1	General aspects of non-LTI modeling . . . . .	131
9.1.1	Concepts . . . . .	131
9.1.2	Port resistance propagation . . . . .	132
9.1.3	Port resistance computation . . . . .	133
9.1.4	Aliasing and frequency warping . . . . .	133
9.1.5	Iteration per sample vs. oversampling . . . . .	134
9.2	Parametric control of WDF elements . . . . .	134
9.2.1	Time-varying resistances . . . . .	134
9.2.2	Time-varying resistive sources . . . . .	135
9.2.3	Time-varying reactances . . . . .	135

9.2.4	Power normalization in time-variant networks . . . . .	139
9.3	Nonlinear resistances . . . . .	141
9.3.1	Parametric iteration of resistance . . . . .	141
9.3.2	Nonlinear resistors with I-port . . . . .	143
9.4	Nonlinear capacitance . . . . .	143
9.4.1	Parametric iteration of a capacitor . . . . .	143
9.5	Nonlinear inductance . . . . .	145
9.5.1	Nonlinear capacitors and inductors with I-port . . . . .	145
9.6	Distributed nonlinearities . . . . .	146
9.7	Non-local interaction and controlled sources . . . . .	146
<b>10</b>	<b>Reduction of Physical Models to DSP Algorithms</b>	<b>147</b>
10.0.1	Reduction of a DWG model to a single delay loop structure . . . . .	147
10.0.2	Commutated DWG synthesis . . . . .	148
10.0.3	Case study: Modelling and synthesis of the acoustic guitar . . . . .	149
10.0.4	DWG modelling of various musical instruments . . . . .	151
10.1	Source-filter models . . . . .	155
10.1.1	Subtractive synthesis in computer music . . . . .	157
10.1.2	Source-filter models in speech synthesis . . . . .	157
10.1.3	Instrument body modelling by digital filters . . . . .	158
10.1.4	The Karplus-Strong algorithm . . . . .	158
10.1.5	Virtual analogue synthesis . . . . .	159
<b>11</b>	<b>Introduction to BlockCompiler (BC)</b>	<b>161</b>
11.1	Basic Lisp syntax . . . . .	162
11.2	Making blocks and patches in BC . . . . .	163
11.2.1	Exporting models to MATLAB/Octave . . . . .	164
11.2.2	Exporting simulation results to MATLAB/Octave . . . . .	166
11.2.3	Real-time simulation in BlockCompiler . . . . .	166
11.2.4	Exporting a patch to Pd external . . . . .	166
11.3	DSP models in BC for MATLAB . . . . .	167
11.3.1	Signal distortion by smooth nonlinearity . . . . .	167
11.3.2	First-order lowpass filter . . . . .	168
11.3.3	Karplus-Strong string synthesis . . . . .	168
11.4	Real-time DSP in BC . . . . .	169
11.4.1	Modulated sine-wave synthesis . . . . .	170
11.4.2	Echo processor . . . . .	171
11.4.3	Real-time Karplus-Strong synthesis . . . . .	172
11.5	Physical modeling in BC . . . . .	172
11.5.1	RC circuit . . . . .	172
11.5.2	Delay line circuit . . . . .	173
11.5.3	Delay line circuit made of unit delay lines . . . . .	174
11.5.4	Definition and use of a macro block . . . . .	175
11.5.5	Simple nonlinearity: Rectification by diode . . . . .	176

<b>12 Physical Modeling Examples</b>	<b>179</b>
12.1 Electric circuits and networks . . . . .	179
12.1.1 RCL circuits . . . . .	179
12.1.2 Transformer circuits . . . . .	182
12.1.3 Transmission lines . . . . .	182
12.1.4 Active analog circuits . . . . .	182
12.1.5 Nonlinear circuits . . . . .	182
12.2 Vibrating systems in mechanics . . . . .	182
12.2.1 Mass-spring systems . . . . .	182
12.2.2 2D systems . . . . .	183
12.3 Acoustic systems . . . . .	183
12.3.1 Lumped systems . . . . .	183
12.3.2 1D-systems . . . . .	183
12.3.3 3D-systems . . . . .	183
12.4 Transducer modeling . . . . .	183
12.5 Loudspeakers . . . . .	183
12.6 Headphones . . . . .	183
12.7 Microphones . . . . .	184
12.8 Modeling and Sound Synthesis of Musical Instruments . . . . .	184
12.9 Hearing? . . . . .	184
12.10 Speech production . . . . .	184
12.11 Etc? . . . . .	184
<b>13 BlockCompiler Programming</b>	<b>185</b>
13.1 Overview of the BlockCompiler . . . . .	186
13.1.1 Current status of the BlockCompiler . . . . .	187
13.1.2 Basic scripting . . . . .	187
13.1.3 Writing macro blocks . . . . .	187
13.1.4 Writing elementary blocks . . . . .	189
13.1.5 Memory management . . . . .	189
13.1.6 Exporting patches . . . . .	190
13.2 Lumped one-port WDF-elements . . . . .	191
13.2.1 WDF resistive elements:Electrical resistor (.R), mechanical damper (.Rm), and acoustical resis	
13.2.2 WDF source elements (1)Voltage source (.E), force source (.Fm), and pressure source (.Pa)19	
13.2.3 WDF source elements (2)Current source (.J), velocity source (.Vm), and flow (volume velocity	
13.2.4 WDF capacitive and compressive elements:Electrical capacitor (.C), mechanical spring (.Cm),	
13.2.5 WDF inductive and inertia elements:Electrical inductor (.L), mechanical mass (.Lm), and acou	
13.2.6 Diode (.diode) . . . . .	194
13.2.7 External root elements (.ext-root) and (.ext-root2) . . . . .	194
13.3 WDF adaptors . . . . .	194
13.3.1 Parallel 3-port adaptor (.par3-adaptor) . . . . .	194
13.3.2 Serial 3-port adaptor (.ser3-adaptor) . . . . .	194
13.3.3 Parallel 2-port adaptor (.par2-adaptor) . . . . .	194
13.3.4 Serial 2-port adaptor (.ser2-adaptor) . . . . .	194
13.4 Two-port lumped WDF-elements . . . . .	195
13.4.1 Transformer (.xformer) . . . . .	195
13.4.2 Gyrator (.gyrator) . . . . .	195



13.5	Two-port distributed WDF-elements . . . . .	195
13.5.1	Delay-line (.dline) . . . . .	195

## **A BC3 Block Library User's Reference 203**

A.1	Manipulation of BlockCompiler patches and objects . . . . .	203
A.1.1	Patch creation . . . . .	203
A.1.2	Patch structure . . . . .	204
A.1.3	Compilation and runtime operations of patches . . . . .	205
A.1.4	Controlling of a running patch . . . . .	206
A.2	Export of BC patches . . . . .	207
A.3	Block classes, functions, and data structures . . . . .	207
A.3.1	Parameters of block creation functions . . . . .	207
A.3.2	Generic keywords . . . . .	208
A.3.3	Runtime data types and objects in BlockCompiler . . . . .	209
A.4	Math functions and operations . . . . .	210
A.4.1	Multi-input arithmetic blocks . . . . .	210
A.4.2	Single input blocks . . . . .	211
A.4.3	Other math blocks . . . . .	212
A.4.4	Predicate blocks for data comparison . . . . .	212
A.4.5	Logic blocks . . . . .	212
A.5	Symbolic operations . . . . .	212
A.6	DSP blocks . . . . .	214
A.6.1	Signal I/O blocks and data items . . . . .	214
A.6.2	Signal generation blocks . . . . .	214
A.6.3	Signal routing . . . . .	215
A.6.4	Digital filter blocks . . . . .	215
A.6.5	Delay blocks . . . . .	219
A.6.6	Miscellaneous DSP blocks . . . . .	220
A.7	WDF elements . . . . .	221
A.7.1	Lumped linear WDF one-port elements with I-port . . . . .	221
A.7.2	Lumped linear WDF one-port elements with T- port . . . . .	222
A.7.3	RLC-combination one-ports . . . . .	222
A.7.4	Symbolic z-expression elements . . . . .	222
A.7.5	Consolidated one-port blocks . . . . .	223
A.7.6	Two-port elements . . . . .	224
A.7.7	Probe elements . . . . .	224
A.8	Connectivity for circuits and networks . . . . .	225
A.9	Circuit synthesis . . . . .	225
A.10	DWG elements . . . . .	225
A.11	FDTD elements . . . . .	225
A.11.1	Misc elements . . . . .	226



# Chapter 1

## Introduction

Computers have changed radically the way of studying and simulating complex physical systems. Particularly numerical simulation is applied to complex cases where analytical calculations are far from feasible. The ever increasing speed and memory size of computers extends constantly the complexity limits of problems that can be solved.

In addition to off-line modeling, computers are able to carry out real-time simulation of complex systems, which enables applications for example in multimedia, communication technology, and virtual reality. In such cases the computational model has to allow real-time output and response to user actions with low enough latency, being from milliseconds up to few hundreds of milliseconds.

This book deals with modeling and numerical simulation of physical systems in the time domain. The methodology for such tasks has evolved over year and includes a multitude of approaches and paradigms have been applied to specific problems in engineering and physical sciences. It is quite common that a single modeling paradigm or maybe two have become popular in a specific problem domain, while others are used in similar cases in another problem domains. It is even less common that several modeling paradigms are combined to solve a single problem. Such hybrid modeling may, however, be advantageous when searching for the most optimal solution for computation. At least it is useful to know the available modeling method and tool palette before tackling with complex modeling problems.

This book started to emerge from material produced in a few research projects, particularly related to physics-based sound source modeling and model-based sound synthesis. For example modeling of musical instruments for real-time sound synthesis requires advanced and highly efficient algorithms. Problems in audio engineering, for example modeling of electro-acoustic transducers (loudspeakers and microphones) bring up similar requirements. In fact, many modeling and simulation tasks applied to various physical systems need roughly equivalent formulations.

In these cases there can be need to work in more than one physical domain, for example combining subsystems of electrical, mechanical, and acoustical components, using multiple paradigms for hybrid modeling, and to find proper computational tools (software) to do this effortlessly enough. It turns out that many popular software tools turn out to be highly inconvenient when trying to apply them to something different from the paradigm they were designed for. Therefore this book provides also a software tool called BlockCompiler (BC), which supports multi-paradigm modeling in multiple physical domains.

The book at hand is intended to serve many needs. Firstly, it tries to bring together different modeling paradigms with a broad view on what theoretical methods and computational means

are available for discrete-time physics-based modeling. It tries to avoid too deep mathematical theories, presenting only the essentials and pointing further to more advanced and detailed literature.

Secondly, the book can be used as a more practical guide with an experimentation approach to physics-based modeling, using the BlockCompiler software. For a non-theoretician and an engineering oriented mind it is often easier first to run practical experiments that provide intuition and qualitative understanding of the phenomena at hand, after which the formal theories also become more transparent and easier to understand.

Because of the two approaches, theoretical and practical, used to write this book, it can be studied both from the beginning to the end in a theory-first way, or from the late chapters with practical examples and simulation and finally return to the theory (if motivation emerged). Some parts of the book may also be so familiar to the reader that he/she can just browse or jump over those parts.

The BlockCompiler software is an essential companion of this publication. It is an object oriented tool for building models, which can then be simulated directly or exported to another software or hardware environment for real-time or non-realtime simulation. BlockCompiler is basically a code generator so that the model structure can be converted to executable code in a specific language or software just by rewriting the code generation part, not touching the model building software itself. A multi-platform version (Windows, Linux, MacOSX) exists (by Dec. 2007), but is only in limited distribution due to continuous changes to the system and unfinished documentation.

## 1.1 General concepts of physics-based modeling

In this section we discuss a number of physical and signal processing concepts that are important in understanding the modeling paradigms discussed in more detail in the subsequent sections. Each paradigm is also characterised briefly in the end of this section.

### 1.1.1 Physical domains, variables, and parameters

Physical phenomena can be categorized to belong in different *physical domains*. The most important ones for our purposes are the *electrical*, the *acoustical* and the *mechanical* domain<sup>1</sup>. Physical domains may have interaction between each other, or they can be used as *analogies* (equivalent models) of each other. Electrical circuits and networks are often applied as analogies to describe systems of the other physical domains.

Quantitative description of a physical system is done through measurable quantities that typically come in pairs of variables, such as force and velocity in the mechanical domain, pressure and volume velocity in the acoustical domain, or voltage and current in the electrical domain. The members of such *dual variable pairs* are categorized here generically as *potential variable* or *across variable*, such as voltage, force, or pressure, and *flow variable* or *through variable*, such as current, velocity, or volume velocity. If there is a linear relationship between the dual variables, this relation can be expressed as a parameter, such as impedance  $Z = U/I$  being the ratio of voltage  $U$  and current  $I$ , or by its inverse, admittance  $Y = I/U$ . An example from the mechanical domain is mobility (mechanical admittance), the ratio of velocity and force. When

---

<sup>1</sup>While electromagnetics is more fundamentally distinct from mechanics, the acoustical domain may be seen more as a special case of the mechanical domain.

using such parameters, only one of the dual variables is needed explicitly, because the other one is achieved through the parametric constraint rule.

The modeling methods discussed in this book use two types of variables for computation: *K-variables* vs. *W-variables*. *K* comes from Kirchhoff and refers to the Kirchhoff continuity rules of quantities in electric circuits and networks [1, 2]. *W* is a shortform for *wave*, referring to wave components of physical variables. Instead of pairs of potential and flow type of *K*-variables, the wave variables come in pairs of *incident* and *reflected* wave components. The details of wave modeling are discussed in Chapters 4 and 6, while *K*-modeling is discussed particularly in Chapter 5. It becomes obvious that these are different representations of the same phenomenon, and the possibility to combine both *K*- and *W*-approaches in hybrid modeling will be discussed in Chapter 7.

The decomposition into wave components is clear in such wave propagation phenomena, where opposite-traveling waves add up to the actual observable *K*-quantities. A wave quantity is directly observable only when there is no opposite-travelling counterpart. It is, however, a highly useful abstraction to apply wave components to any physical cases, since this helps in solving computability (causality) problems in discrete-time modeling.

### 1.1.2 Physical structure and interaction

Physical phenomena are observed as structures and processes in space and time. If there are quantities in a system that change in time, i.e., they are variables, it is called a *dynamic* system, otherwise it is a *static* one. Constant quantities and especially slowly varying controllable quantities in dynamic systems are called parameters. Here we are primarily interested in modeling of dynamic behaviour. As a universal property in physics, the interaction of entities in space always propagates with a finite velocity. This may have orders of magnitude differences in different physical domains, the speed of light being the ultimate limit.

*Causality* is a fundamental physical property that follows from the finite velocity of interaction propagating from a cause to the corresponding effect. Many relations between observed quantities in physical systems are not causal. For example the relation of voltage and current through impedance as given above is only a constraint and the variables can be solved only in a context of the whole circuit. The requirement of causality introduces special computability problems in discrete-time simulation, because a two-way interaction with no delay leads to the *delay-free loop problem*. In such case the value of a variable is needed for the computation of other variables before the value of the variable at hand is known. Mathematically this is called an implicit equation. The use of wave variables helps in this sense, because the incident and reflected waves are in causal relationship. Particularly the wave digital filters, discussed in Chapter 6, take a careful treatment of this problem through the use of *W*-variables and specific scheduling of computation operations.

Taking the finite propagation speed into account requires using a spatially *distributed* model. Depending on the case at hand, this can be a full 3-D (three-dimensional) model, such as in room acoustics, a 2-D model, for example as for a drum membrane, or a 1-D model, such as for a vibrating string. If the object to be modeled behaves homogeneously as a whole, for example due to its small size compared to the wavelength of wave propagation, it can be represented by a *lumped* model that does not need spatial dimensions.

In the physical macro-scale of interest here, space and time can be considered inherently continuous. Differential equations, such as the wave equation, are perfectly suited to characterize such continuity. Numerical computation cannot utilize this continuity, because systems to be

modelled must be discretized, i.e., quantized to a regular space-time grid or to discrete elements and structures in time and space. This is an approximation, a compromise between accuracy requirements and manageable complexity. Therefore a wise selection of data representations and modeling methods is important in order to obtain optimal or at least useful results.

### 1.1.3 Signals, signal processing, and discrete-time modeling

The word *signal* means typically the value of a measurable or observable quantity as a function of time and possibly as a function of place. In signal processing, signal relationships typically represent one-directional cause-effect chains. It can be achieved technically by active electronic components in analog signal processing or by numerical computation in digital signal processing (DSP). This simplifies the design of circuits and algorithms compared to two-way interaction that is common in (passive) physical systems, for example in systems where the reciprocity principle of cause and effect is valid. In true physics-based modeling the two-way interactions must be taken into account. This means that from the signal processing viewpoint such models are full of feedback loops, which further implicates that the concepts of computability (causality) and stability become crucial, as will be discussed later below.

In this book we apply the discrete-time signal processing approach to physics-based modeling whenever possible. The motivation to this is that digital signal processing is an advanced theory and tool emphasizing computational issues, particularly maximal efficiency. This is crucial particularly in real-time simulation and sound synthesis. Signal flow diagrams are also a good graphical means to illustrate algorithms underlying the simulations.

In discrete-time numerical modeling and simulation based on digital signal processing the system under study must be discretized, i.e., the variables have to be sampled in proper time intervals and spatial locations. In digital computers and processors the samples need further be quantized in amplitude scale for numerical representation with finite precision. While the numerical accuracy issues may be important in particular implementations, the standard floating-point formats available make this less interesting from a systems point of view, and will not gain much attention in this book.

The concepts of sampling rate and spatial sampling resolution need some focus in this context. According to the sampling theorem [3], signals must be sampled so that at least two samples have to be taken per period or wavelength for sinusoidal signal components or their combinations in order to make perfect reconstruction of a continuous-time signal possible. This limit frequency, one half of the sampling rate, is called the Nyquist frequency. If a signal component higher in frequency, say  $f_x$ , is sampled by rate  $f_s$ , it will be *aliased*, i.e., mirrored by Nyquist frequency back to the *base band* below the Nyquist frequency by  $f_a = f_s - f_x$ . In audio signals this will be perceived as very disturbing distortion, and should be avoided. In linear systems (see the next subsection below, if the inputs are bandlimited properly, aliasing is not a problem because no new frequency components can be created, but in nonlinear systems aliasing is problematic. In modeling physical systems it is important to remember also that *spatial aliasing* can be a problem if the spatial sampling grid is not dense enough compared to wavelength.

### 1.1.4 Linearity and time-invariance

*Linearity* of a system means that the superposition principle is valid, i.e., quantities and signals in a system behave additively without ‘disturbing’ each other. A linear system cannot create any

signal components with new frequencies. If a system is nonlinear, it typically creates harmonic (integer multiples) or intermodulation (sum and difference) frequency components. This is particularly problematic in discrete-time computation because of the aliasing of new signal frequencies beyond the Nyquist frequency.

If a dynamic system is both *linear and time-invariant* (LTI), there are constant-valued parameters that effectively characterize its behaviour. We may think that in a time-varying system its characteristics (parameter values) change according to some external influence, while in a nonlinear system the characteristics change according to the signal values within the system.

Linear systems or models have many desirable properties. In digital signal processing, LTI systems are not only easier to design but also typically more efficient and robust computationally. A linear system can be mapped to transform domains where the behaviour can be analyzed by algebraic equations [4]. For continuous-time systems the Laplace and Fourier transforms (see Section ??) can be applied to map between the time and frequency domains, and the Sturm-Liouville transform [5] applies similarly to the spatial dimension. For discrete-time systems the  $z$ -transform and the discrete Fourier transform (DFT and its fast algorithm, FFT) (Section ??) are used.

For nonlinear systems there is no such elegant theory as for the linear ones, rather there are many forms of nonlinearity, requiring different ways to deal with them. In discrete-time modeling, nonlinearities bring problems that are difficult to solve. In addition to aliasing that was already mentioned, the delay-free loop problem and stability problems become worse than for linear systems. If the nonlinearities in a system to be modeled are spatially distributed, the modeling task is even more difficult than with a localized nonlinearity. Nonlinearities will be discussed in several parts of this book, but particularly in Chapter 9.

### 1.1.5 Energetic behaviour and stability

The product of dual variables such as voltage and current gives the power, which, when integrated in time, yields the energy. Conservation of energy in a closed system is a fundamental law of physics that should be obeyed also in true physics-based modeling.

A physical system can be considered *passive* or *inexpensive* in energetic sense if it does not produce energy, i.e., it preserves its energy or dissipates it into another (such as thermal) energy form. For example in musical instruments the resonators are typically passive, while the excitation (plucking, bowing, blowing, etc.) is an *active* process that injects energy to the passive resonators.

The *stability* of a physical system is closely related to its energetic behaviour. Stability can be defined so that the energy of the system remains finite for finite-energy excitations. In this sense a passive system always remains stable. From the signal processing viewpoint, stability may also be meaningfully defined so that the variables, such as voltages, remain within a linear operating range for possible inputs in order to avoid signal clipping and distortion. For system transfer functions, stability is typically defined so that the system poles (roots of the denominator polynomial) in the Laplace transform remain in the left half plane, or that the poles in the  $z$ -transform in a discrete-time system remain inside the unit circle [4]. This guarantees that there are no responses growing without bounds for finite excitations.

In signal processing systems with one-directional interaction between stable subblocks, instability can appear only if there are feedback loops. In general it is impossible to analyze if such a system is stable or not without knowing the whole feedback structure. Contrary to this, in models with physical two-way interaction the passivity requirement is a sufficient condition

of stability, i.e., if each element is passive, then any arbitrary network of such elements remains stable.

### 1.1.6 Modularity and locality of computation

For a computational realization it is desirable to decompose a model systematically into blocks and interconnections between them. Such an object-oriented or block-based approach helps in managing with complex models through the use of the modularity principle. The basic modules can be formulated to correspond to elementary objects or functions in the physical domain at hand. Abstractions by defining new more complex macro blocks on the basis of more elementary ones helps hiding details when building excessively complex models.

For one-directional interaction in signal processing it is enough to have input and output terminals for connecting of blocks. For physical interaction the connections need to be done through ports, each port having a pair of K- or W-variables depending on the modeling paradigm used. This follows the mathematical principles used for electrical networks [2]. Details on the block-wise construction of models will be discussed in the following sections for each modeling paradigm.

Locality of interaction is a desirable modeling feature, which is also related to the concept of causality. In a physical system with a single propagation speed of waves it is enough that a block interacts only with its nearest neighbours and does not need global connections to compute its task. If the properties of one block in such a strictly localized model vary, the effect automatically propagates throughout the system. On the other hand, if some effects propagate for example with the speed of light but others with the speed of sound in the air, the former effect is practically simultaneously everywhere. If the sampling rate in a discrete-time model is tuned to audio bandwidth (typically 44.1 or 48 kHz sample rate), the unit delay between samples is all too long to represent light wave propagation between blocks. Two-way interaction with zero delay means a delay-free loop, the problem that we face several times in this book. Technically it is possible to realize fractional delays [6], but delays shorter than the unit delay contain a delay-free component, so the problem is hard to avoid. There are ways to make such systems computable, but the cost in time (or accuracy) may become prohibitive at least for real-time processing.

### 1.1.7 Complexity in physics-based modeling

Models always only approximate real physical phenomena. Therefore they reduce the complexity of the target system. This may be intentional for example to keep the conceptual complexity or computational cost manageable, or more generally to keep some cost function below an allowed limit. Particularly important these constraints are in real-time sound synthesis and simulation. Limitations in complexity of a model are often needed because the target system is conceptually overcomplex to a scientist or engineer developing the model, and thus cannot be improved within the competence or effort available. An overcomplex system may be deterministic and modellable in principle but not in practice, it may be stochastic due to noise-like signal components, or it can be chaotic so that infinitesimally small disturbances lead to unpredictable states. In this book we deal with deterministic models in the first hand; statistically behaving elements in the models appear only rarely.

Although this book is about physics-based modeling, we must remember that in many applications an important form of complexity to be taken into account is perceptual (over)complexity.



For example in sound synthesis there may be no reason to make the model more precise, because listeners cannot hear the difference. Phenomena that are physically prominent but do not have an audible effect can be excluded from models in such cases.

## 1.2 Overview of paradigms for physics-based discrete-time modeling

The rest of this book concentrates on physics-based methods and techniques of discrete-time modeling. We only briefly mention several methods that can be very useful in frequency-domain modeling but do not easily solve the task of time-domain simulation, particularly in real-time synthesis. For example methods to solve the underlying partial differential equations are theoretically important but do not directly help in simulation and synthesis. Finite element and boundary element methods are generic and powerful in solving system behavior numerically, particularly for linear systems in the frequency domain, but we focus on inherently time domain methods. Three-dimensional spaces, such as rooms and enclosures, can be modelled by the image source and ray tracing methods combined with late reverberation algorithms, but they can be seen as semi-physical models due to lack of two-directional interaction of physical elements.

The main paradigms in discrete-time modeling can be briefly characterized as follows.

### 1.2.1 Modeling paradigms

#### THE FOLLOWING NEEDS TO BE UPDATED

##### Digital waveguides

in Section ?? are the most popular physics-based method of modeling and synthesizing musical instruments based on 1-D resonators, such as strings and wind instruments. The reason to this is their extreme computational efficiency in their basic formulations. They have been used also in 2-D and 3-D modeling, but in such cases they are not superior in efficiency anymore. Digital waveguides are based on the use of wave components, thus they form a wave modeling (W-modeling) paradigm. Therefore they are also compatible to wave digital filters (Section ??), but they need special conversion techniques to be compatible with K-modeling techniques in order to construct hybrid models as discussed in Section ??.

##### Finite difference models

in Section ?? are the numerical replacement to solving partial differential equations. Differentials are approximated by finite differences so that time and position will be discretized. By proper selections of discretization to regular meshes the computational algorithms become simple and relatively efficient. Finite difference time domain (FDTD) schemes are K-modeling methods, since wave components are not explicitly utilized in computation. FDTD schemes have been applied successfully to 1-D, 2-D, and 3-D systems, although in linear 1-D cases digital waveguides are typically superior in computational efficiency and robustness. In multidimensional mesh structures the FDTD approach is more efficient. It also shows potential to

deal systematically with nonlinearities (see Section ??). FDTD algorithms can be problematic due to lack of numerical robustness and stability, unless carefully designed.

### Wave digital filters

in Section ?? are another wave-based modeling technique, originally developed for discrete-time simulation of analog electric circuits and networks. In their original form wave digital filters are best suited for lumped element modeling, thus they can be easily applied to wave-based mass-spring modeling. Due to their compatibility with digital waveguides, these methods are complementing each other. Wave digital filters have also been generalized to multidimensional networks and to systematic and energetically consistent modeling of nonlinearities. They have been applied particularly to deal with lumped and nonlinear elements in models where spatially distributed parts are typically realized by digital waveguides.

### Modal decomposition methods

in Section ?? represent another perspective to look at vibrating systems, conceptually from a frequency domain viewpoint. The eigenmodes of a linear system are exponentially decaying sinusoids at eigenfrequencies in the response of a system to impulse excitation. Although the thinking by modes is related to the frequency domain, simulation by modal methods can be relatively efficient, and therefore suitable to discrete-time computation. Modal decomposition methods are inherently based on the use of K-variables. Modal synthesis has been applied to make convincing sound synthesis of different musical instruments. Functional transformation method (FTM) is a recent development of systematically exploiting the idea of spatially distributed modal behavior, and it has been extended also to nonlinear system modeling.

### Mass-spring networks

in Section ?? are an intuitively easy to comprehend modeling approach, where the basic elements in mechanics, i.e., masses, springs, and damping elements, are used to construct vibrating structures. It is inherently a K-modeling methodology, which has been used to construct small- and large-scale mesh-like and other structures. It has close relations to FDTD schemes in mesh structures and to wave digital filters for lumped element modeling. Mass-spring networks can be realized systematically also by wave digital filters using wave variables (Section ??).

### Source-filter models, NOT REALLY PHYSICAL

in Section 10.1 form a paradigm between physics-based modeling and signal processing. The true spatial structure is not visible anymore, but is transformed into a transfer function that can be realized as a digital filter. The approach is attractive in sound synthesis because digital filters are optimized to implement transfer functions. The source part of a source-filter model is often a wavetable, consolidating different physical or synthetic signal components needed to feed the filter part. The source-filter paradigm is often used in combination with other modeling paradigms in more or less ad hoc ways.

### 1.2.2 Modeling typology

The physics-based modeling paradigms overviewed above can be categorized according to the main principles they are based on. Although such typology is somewhat violent to the modeling capabilities of these paradigms, the categorization may help in understanding the main features and strengths of them. The following table is one possible attempt to put structure to the field of physics-based modeling paradigms.

Table 1.1: Typology for physics-based modeling paradigms.

	Kirchhoff (K) vs. Wave (W)	Distributed (D) vs. Lumped (L)	Derivation domain: Time (T) vs. Freq. (F)
DWG	W	D (L)	T
FDTD	K	D	T
WDF	W	L (D)	T
Modal/FTM	K	D/L	F
Z/Y (filter)	K	L	F/T
Mass/Spring	K	L	T

The categorization is done in three dimensions:

1. Kirchhoff (K) vs. Wave (W) modeling
2. Distributed (D) vs. Lumped (L) modeling
3. Derivation domain (Time vs. Frequency)

The first and second dimensions (K vs. W and D vs. L) are as described earlier. The derivation domain, Time (T) vs. Frequency (F), means here which way of thinking or analysis is used to derive the elements applied in each modeling paradigm. While in this book we are focusing on on discrete-time modeling so that the algorithms are finally computed in the time domain, some paradigms may have been derived or designed also in the frequency domain.



# Chapter 6

## *Lumped W-Modeling:* **Wave Digital Filter (WDF) Models**

The theory of *wave digital filters* (WDF) emerged as a systematic approach to discrete-time approximation of analog circuits and networks, as proposed by Fettweis [58]. Originally it was developed primarily for the simulation of lumped electric components and circuits, although elements for spatially distributed systems were available and extensions to multidimensional systems modeling have since then been included [57]. In the framework of filter design the approach is based on selecting analog reference filters and to map them to corresponding discrete-time filters. WDFs have also been applied to other physical domains than electric circuits and networks. Our interest here is not as much filter design as to model and simulate physical systems.

Wave digital filters have several attractive features. They have a conceptually clear correspondence to the elements and building blocks of the physical world. Computation with WDFs is modular and localized, particularly for linear systems, and the temporal causality of cause and effect is explicit and helps in understanding realizability issues. Nonlinearities, always being complicated in modeling and simulation, find some interesting solutions in WDFs. Furthermore, and as one of their strongest features, WDFs have a very clear formulation in terms of passivity, losslessness, and stability, whereby the energetic behavior (e.g., stability) of an arbitrary network can be guaranteed through the properties of each element locally, particularly for time-invariant cases. Another important advantage of WDFs is their numerical robustness, so that the stability in finite word length simulation remains controlled and predictable.

A relatively rich literature basis exists on WDFs and their wide range of applications, from electric circuits to general relativity, see for example [58, 57, 35, 59, 60, 61, 62, 63, 64].

### 6.1 WDF theory

WDF models are based on the use of two-directional interaction between ports of circuit elements using wave variables. According to the original notation of Fettweis [58], the incident wave  $a$  and the reflected wave  $b$  at a port are defined as a linear mapping from electric K-variables, i.e., voltage  $u$  and current  $i$ , and the inverse mapping as<sup>1</sup>

---

<sup>1</sup>This formulation declares voltage waves. Another choice is to use current waves by writing  $a = Gu + i$  and  $b = Gu - i$ , where  $G$  is port conductance. This is an equivalent formulation and does not bring anything new to using voltage waves.

$$\begin{cases} a = u + Ri \\ b = u - Ri \end{cases} \Leftrightarrow \begin{cases} u = (a + b)/2 \\ i = (a - b)/2R \end{cases} \quad (6.1)$$

where  $R$  is a real-valued computational parameter called *port resistance*. Figure 6.1 depicts the port variables at a port of a circuit element, presented in K- and W-variables. By proper selection of port resistance  $R$  and the mapping from  $a$  to  $b$  inside the element, computationally elegant and advantageous simulation of circuits, interconnected through ports, can be obtained.



Figure 6.1: Definition of (a) K- and (b) W-variables at a port of a circuit element.

One of the advantages of the W-variable formulation over the use of K-variables is physical *causality* in W-variable computation  $b = f(a)$ . While K-variables at a port do not have a causal relationship, i.e., voltage cannot be solved from current or vice versa without knowing the rest of the circuit the port is connected to, in W-variables  $b$  depends only on  $a$  through the inner functioning of the circuit element. Physical causality also means that the response  $b$  follows in time the excitation  $a$ . Furthermore, causality implies *localization* of computation. By proper selection of port resistances and element operations, delay-free loops that appear when trying to use K-variables can be avoided. One more advantage of wave-based formulations is their numerical robustness [58, 35].

### 6.1.1 Physically normalized waves

In this presentation we will slightly modify the definition of W-variables versus K-variables in order to make WDFs fully compatible with the digital waveguides that were discussed in Chapter 4. Now we denote voltage by  $U$ , current by  $I$ , port resistance by  $R_p$ , and wave components by  $A$  and  $B$ . Capital letters denote that the variables (except  $R_p$ ) are considered as  $z$ -transform expressions. Let us first generalize the WDF definition of Eq. (6.1) by writing

$$\begin{cases} A = k(U + R_p I) \\ B = k(U - R_p I) \end{cases} \Leftrightarrow \begin{cases} U = (A + B)/2k \\ I = (A - B)/2kR_p \end{cases} \quad (6.2)$$

i.e., a real-valued scaling factor  $k$  is included. Factor  $k$  does not change the behavior of the port in terms of K-variables or W-variables, it only changes the scaling between K- and W-variable pairs. Therefore we can select any finite value of  $k$  as far as we are consistent in the interpretation of the variables.

In a closer look at the DWGs in Section 4.1.4, Eq. (4.12), we can find that a physically motivated interpretation of the wave variables requires that  $A = P_i^+$ ,  $B = P_i^-$ , and  $P = A + B$ . In the electrical domain this means that  $A = U^+$ ,  $B = U^-$ , and  $U = A + B$ . Therefore a

physically appropriate scaling of waves is obtained by  $k = 1/2$ . In the electrical domain the definition becomes

$$\begin{cases} A = (U + R_p I)/2 \\ B = (U - R_p I)/2 \end{cases} \Leftrightarrow \begin{cases} U = (A + B) \\ I = (A - B)/R_p \end{cases} \quad (6.3)$$

This formulation of *physically normalized waves* will be used here as the basic definition for WDFs, making WDF ports directly compatible with DWG ports without scaling by a factor that is otherwise needed. A computational advantage obtained also is that conversion from W-variables (primary interest for computation) to K-variables (primary interest for observation) does not need the coefficient  $k = 1/2$  as is needed when using Eq. (6.1).

The power flowing through a WDF port with physically normalized waves is

$$W = V \cdot I = (A + B) \frac{(A - B)}{R_p} = \frac{A^2}{R_p} - \frac{B^2}{R_p} \quad (6.4)$$

which shows that  $a$  brings in  $W = A^2/R_p$  and  $b$  brings out  $W = B^2/R_p$ . If the transfer function  $H(z) = B(z)/A(z)$  from  $A$  to  $B$  is an allpass function with unity value of magnitude response, the port is lossless (energy-preserving), since the energy flowing in also flows out (within some delay).

### 6.1.2 Power-normalized waves

Another useful convention for the scaling factor  $k$  is to select  $k = 1/(2\sqrt{R_p})$ , whereby

$$\begin{cases} \tilde{A} = (U + R_p I)/(2\sqrt{R_p}) \\ \tilde{B} = (U - R_p I)/(2\sqrt{R_p}) \end{cases} \Leftrightarrow \begin{cases} U = (\tilde{A} + \tilde{B})\sqrt{R_p} \\ I = (\tilde{A} - \tilde{B})/\sqrt{R_p} \end{cases} \quad (6.5)$$

Now the power  $W$  flowing through a power-normalized port is

$$W = \tilde{A}^2 - \tilde{B}^2 \quad (6.6)$$

i.e., it is independent of the port resistance  $R_p$ . The waves in this case are called *power-normalized waves* [35, 57]. The advantage of this choice is that the port power is independent of port resistance  $R_p$ . Power-normalization is discussed further in Section 9.2.4.

### 6.1.3 Connectivity of wave ports

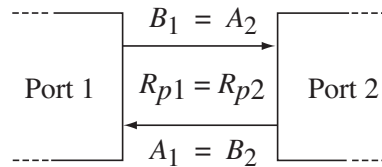


Figure 6.2: Interconnection of two ports.

Two ports are compatible and can be connected together as shown in Fig. 6.2 if they meet the following requirements:

1. They correspond to the same physical domain (such as electrical, mechanical, or acoustical domain). Connection of ports from different domains need transducers elements between them, see Section 6.7, or the subsystems must first be mapped to the same physical domain.
2. Variable types must be the same, for example both must be W-variables or K-variables. A KW-converter is needed to connect a W-port to a K-port or the K-type subsystem must first be modified somehow into a W-subsystem. Additionally, for W-variables the normalization factor  $k$  used in Eq. (6.3) must be the same for both ports.
3. Port resistance values must be the same. If they are not equal, an adaptor (Section 6.4) or a transformer is needed for connection.

Other restrictions may also apply, such as avoidance of delay-free loops. The interconnectivity is discussed particularly in Section 6.4.

## 6.2 Basic WDF elements

This section presents formulations for the most basic WDF elements, i.e., one-port lumped elements, including resistive, capacitive, and inductive elements, as well as voltage and current sources. The formulations are readily applicable in other physical domains through the analogy principles presented in Section ??.

The derivations of WDF elements are based on using  $z$ -transform notation for K- and W-variables, which for the physically normalized wave is written as

$$\begin{cases} U(z) = A(z) + B(z) \\ I(z) = \{A(z) - B(z)\}/R_p \end{cases} \quad (6.7)$$

Now the impedance  $Z(z)$  observable at the port is

$$Z(z) = \frac{U(z)}{I(z)} = R_p \frac{A(z) + B(z)}{A(z) - B(z)} = R_p \frac{1 + B(z)/A(z)}{1 - B(z)/A(z)} \quad (6.8)$$

When denoting the transfer function  $B(z)/A(z)$  by  $H(z)$  this becomes

$$Z(z) = R_p \frac{1 + H(z)}{1 - H(z)} \quad (6.9)$$

$H(z)$  is called the *reflectance* because it is the relation of the reflected and incident waves. By selecting  $H(z)$  properly in Eq. (6.9), different continuous-time (analog) elements can be approximated by discrete-time models as discussed in this section. If  $Z(z)$  is given as a target impedance, the corresponding  $H(z)$  is solved as

$$H(z) = \frac{Z(z) - R_p}{Z(z) + R_p} \quad (6.10)$$

There are, however, some restrictions to be taken into account. For computability reasons we are primarily interested in elements that do not have a delay-free reflection, i.e., the impulse response  $h(n)$  corresponding to  $H(z)$  must have  $h(0) = 0$ . We also require that the port resistance is dependent on the element properties only, not on the other port it is connected to. Such a port is called here a *T-port*, see Section 6.4.

Elements with delay-free reflection in port transfer function  $H(z)$  can be useful, too. They will be discussed in Section 6.3.



### 6.2.1 Resistance

An ideal analog *resistor* is a lossy (energy consuming) and linear element, for which

$$U = R I \quad (6.11)$$

where  $U$  is the voltage across the port,  $I$  is the current through the port, and  $R$  is the resistance value.

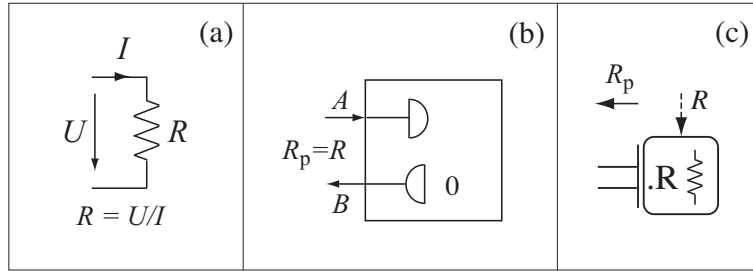


Figure 6.3: Resistance: (a) electrical resistor, (b) WDF resistor, where  $A$  is terminated and not used in computation, and (c) resistor symbol used for BlockCompiler. Data flow direction of the port resistance  $R_p$  is also shown as well as possible parametric control of resistance  $R$ .

A WDF resistor  $R$  is formed simply by selecting  $H = 0$  in Eq. (6.9) in order to obtain a totally reflection-free port so that

$$Z_R = R = R_p \quad (6.12)$$

Such a resistor has an exact correspondence to the ideal analog resistor for the full frequency range  $0 \leq f \leq f_s$ , where  $f_s$  is the sample rate. Notice that now for a WDF resistor of Eq. (6.12)  $B = U - R_p I = R I - R_p I = 0$  (zero reflectance), and therefore  $A$  is not needed in computation at all. The power flowing through a WDF resistor port is

$$W_R = UI = A^2/R_p \quad (6.13)$$

which means that it is always positive for any input different from zero, and thus the resistive element is passive and lossy (dissipates energy).

In the mechanical domain the corresponding resistive element is a damper with mechanical resistance  $R_m$ , which is defined as the ratio of force  $F$  and velocity  $V$ , i.e.,  $R_m = F/V$ . In the acoustical domain, respectively, the acoustical resistance  $R_a = P/Q$ , where  $P$  is pressure and  $Q$  is volume velocity.

Figure 6.3 shows the electrical symbol and WDF realization symbol for a resistor. A resistor with a I-port is derived in Section 6.3.1, realization of nonlinear resistors is discussed in Section ??, and Appendix 13.2.1 presents the use of resistive elements in BlockCompiler.

### 6.2.2 Conductance

Since *conductance* is the inverse of resistance,  $G = 1/R$ , the formulation of WDF resistance is valid also for the WDF conductance as far as the port resistance  $R_p = 1/G$ , where  $G$  is the desired conductance. The same holds in the mechanical and acoustical domains as well. The usage and implementation of WDF conductances in BlockCompiler is presented in 13.2.1.

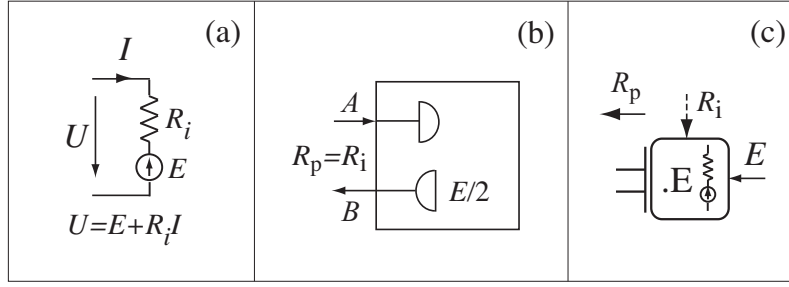


Figure 6.4: Voltage source: (a) electrical symbol, (b) WDF principle, and (c) symbol used for BlockCompiler.

### 6.2.3 Voltage source

The *voltage source* (Fig. 6.4) is a controlled (or constant-valued) source.

The function of a voltage source can be derived from the fact that

$$U = E + R_i I \quad (6.14)$$

where  $R_i$  is the internal resistance and  $E$  the open-circuit voltage of the source, by

$$\begin{cases} A + B = E + R_i I \\ A - B = R_p I \end{cases} \quad (6.15)$$

When selecting  $R_p = R_i$ , we get

$$B = E/2 \quad (6.16)$$

i.e., the reflected wave  $B$  depends only on the source voltage  $E$ . The power flowing through the port is

$$W_E = UI = \frac{A^2}{R_p} - \frac{E^2}{4R_p} \quad (6.17)$$

where the latter term means the power supplied by the source out from the port.

In the mechanical and acoustical domains the corresponding sources are the force source and the pressure source, respectively.

Figure 6.4 shows the electrical symbol of a voltage source and its WDF realization. Section ?? presents the use of voltage, force, and pressure source elements in BlockCompiler.

### 6.2.4 Current source

The *current source* (Fig. 6.5) is another example of controlled (or constant) sources, in this case for the flow variable.

A WDF principle for it can be derived from

$$U = R_p J + R_i I \quad (6.18)$$

where  $R_i$  is the internal resistance and  $J$  the short-circuit current of the source, as

$$\begin{cases} A + B = R_p J + R_i I \\ A - B = R_p I \end{cases} \quad (6.19)$$

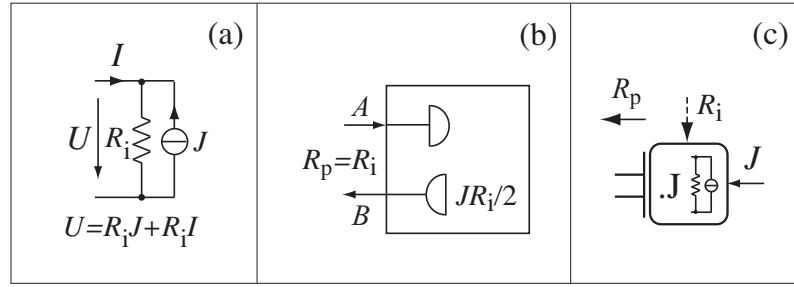


Figure 6.5: Current source: (a) electrical symbol, (b) WDF principle, and (c) symbol used for BlockCompiler.

Selecting  $R_p = R_i$  yields

$$B = R_p J / 2 \quad (6.20)$$

i.e., the reflected wave  $B$  depends on the source current  $J$  and internal resistance  $R_i$ . The power flowing through the port of the current source is

$$W_J = UI = \frac{A^2}{R_p} - \frac{R_p J^2}{4} \quad (6.21)$$

In the mechanical and acoustical domains the sources corresponding to the current source are the velocity source and the volume velocity source, respectively.

Figure 6.5 shows the electric symbols used for a current source and its WDF implementation. Section ?? presents the use of current source elements in BlockCompiler.

### 6.2.5 Capacitance

An ideal analog *capacitor* (Fig. 6.6) is a lossless (energy preserving) reactive element, for which

$$i(t) = C \frac{du(t)}{dt} \quad (6.22)$$

i.e., current  $i(t)$  is the time derivative of voltage  $u(t)$  times capacitance  $C$ .

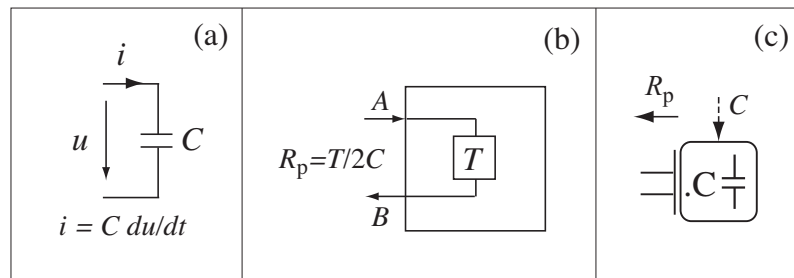


Figure 6.6: Capacitance: (a) electrical capacitor, (b) WDF capacitor, and (c) symbol used for BlockCompiler.

Based on the Laplace transform (Section ??), the impedance of a capacitance is  $Z_C(s) = 1/(sC)$ . When applying the bilinear mapping in Eq. (3.27) we obtain the discrete-time form

$$Z_C(z) = \frac{T}{2C} \cdot \frac{1 + z^{-1}}{1 - z^{-1}} \quad (6.23)$$

where  $T$  is the sample period. By comparing with Eq. (6.9) we notice that a simple solution, which also avoids the delay-free loop problem, is to select<sup>2</sup>

$$R_p = \frac{T}{2C} = \frac{1}{2f_s C} \quad (6.24)$$

$$H(z) = z^{-1} \quad (6.25)$$

$$Z_C(z) = R_p \frac{1 + z^{-1}}{1 - z^{-1}} \quad (6.26)$$

Another elegant way to derive the WDF capacitance equations is to start from the integral equation version of Eq. (6.22)

$$u(t) = u(t_0) + \frac{1}{C} \int_{t_0}^t i(\tau) d\tau \quad (6.27)$$

This can be discretized in time by the trapezoidal rule of numerical integration as

$$u(k) \approx u(k-1) + \frac{1}{C} \cdot \frac{i(k) + i(k-1)}{2} T \quad (6.28)$$

where  $T$  is the sample period and  $k$  is time index of such samples. By rearranging terms and applying the  $z$ -transformation we get

$$U(z) - \frac{T}{2C} I(z) = \left( U(z) + \frac{T}{2C} I(z) \right) z^{-1} \quad (6.29)$$

Now comparing with Eq. (6.3) it is easy to see that by selecting  $R_p = T/(2C)$  we get  $2B(z) = 2A(z)z^{-1}$  and finally end up with Equations (6.25) and (6.26).

The behavior of a WDF capacitance at low frequencies can be analyzed by inserting  $z^{-1} = e^{-j\omega T}$  in Eq. (6.26) so that

$$Z_C(j\omega) = R_p \frac{1 + e^{-j\omega T}}{1 - e^{-j\omega T}} \approx R_p \frac{1 + (1 - j\omega T)}{1 - (1 - j\omega T)} \approx \frac{2R_p}{j\omega T} \quad (6.30)$$

which corresponds to the analog capacitance  $Z(j\omega) = 1/j\omega C$  when

$$R_p = T/(2C) \quad (6.31)$$

as was chosen in Eq. (6.24) above. The impedance of a bilinearly mapped WDF capacitor as a function of frequency is compared against an analog capacitance behavior in Fig. 6.7, and the error percentage is shown in Fig. 6.8. The error is due to *frequency warping* at high frequencies, Eq. (3.29), whereby the analog frequency  $f_a \rightarrow \infty$  when digital frequency goes to the Nyquist frequency  $f_d \rightarrow f_s/2$ . The nature of the frequency warping inherits from the characteristics of the bilinear conformal mapping

$$s = \alpha + j\omega = \frac{2}{T} \frac{z - 1}{z + 1} \rightarrow \omega = \text{Re}\left(\frac{2}{T} \frac{e^{-j\Omega T} - 1}{e^{-j\Omega T} + 1}\right) = \frac{2}{T} \tan\left(\frac{\Omega T}{2}\right) \quad (6.32)$$

where  $\omega$  is the angular frequency in the continuous-time system and  $\Omega$  is the corresponding warped frequency in the bilinearly mapped WDF domain. The frequency warping error can be alleviated by oversampling or by prewarping using Eq. 6.32, see example in ??.

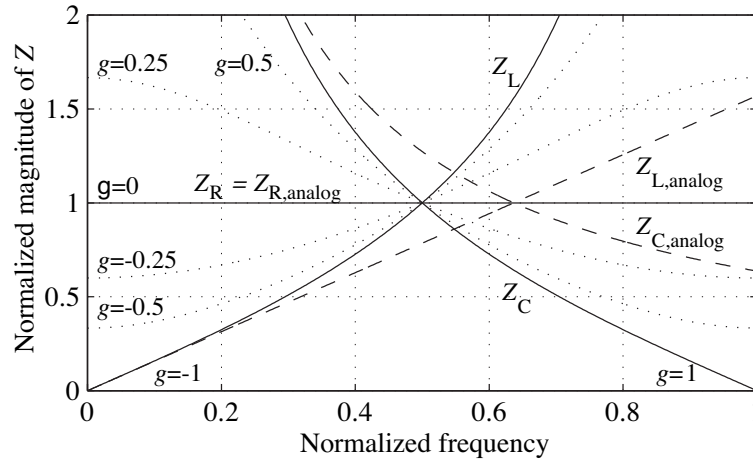


Figure 6.7: Normalized impedance  $|Z/R_p|$  for WDF elements:  $Z_C$  for capacitance,  $Z_L$  for inductance,  $Z_R$  for resistance. Impedances of corresponding analog capacitance ( $Z_{C,analog}$ ) and analog inductance ( $Z_{L,analog}$ ) are plotted by dashed lines. Frequency scale is normalized to Nyquist frequency  $f_s/2$ . Generalized first order reflectance cases with delay feedback coefficient  $g = -1 \dots 1$  are plotted in dotted lines, as discussed in Section 6.2.7.

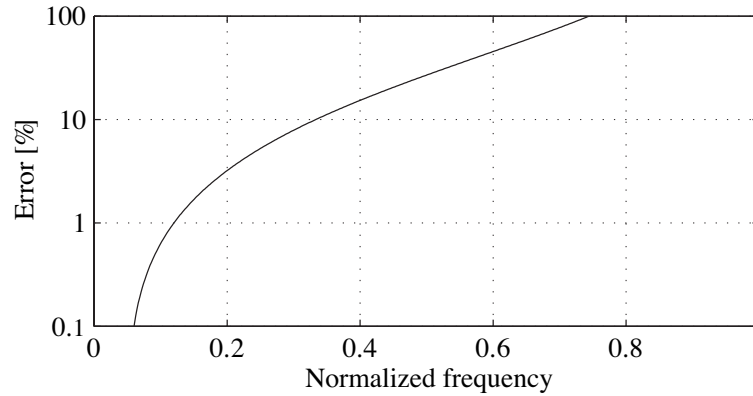


Figure 6.8: Error percentage of WDF inductor impedance as a function of normalized frequency compared to analog inductance. Error is computed as  $e = 100\% * |(Z_L/Z_{L,analog}) - 1|$ . The same error curve holds for capacitors when computed by  $e = 100\% * |(Z_{C,analog}/Z_C) - 1|$ . By an oversampling factor of 8 the error remains within about 1 % in the base band.

Because  $H(z) = z^{-1}$  is an allpass function, the WDF capacitance is a lossless element that works as an energy storage, see Eq. (6.4).

In the mechanical and acoustical domains the corresponding elements are compliance (mechanical capacitor) and acoustic cavity (acoustic capacitance), respectively.

For BlockCompiler usage and implementation of capacitances, see ??.

<sup>2</sup>It can be shown that this particular discrete-time approximation of an analog capacitance is the best lossless approximation if the low-frequency behavior needs to be optimized, as is often the case in audio applications. Other choices of  $H(z)$  may be more optimal for other ranges of the frequency band.

### 6.2.6 Inductance

An ideal analog *inductor* (Fig. 6.9) is a lossless (energy preserving) reactive element, for which

$$u(t) = L \frac{di(t)}{dt} \quad (6.33)$$

i.e., the port voltage  $u(t)$  is proportional to the derivative of current  $i(t)$  times the inductance  $L$ .

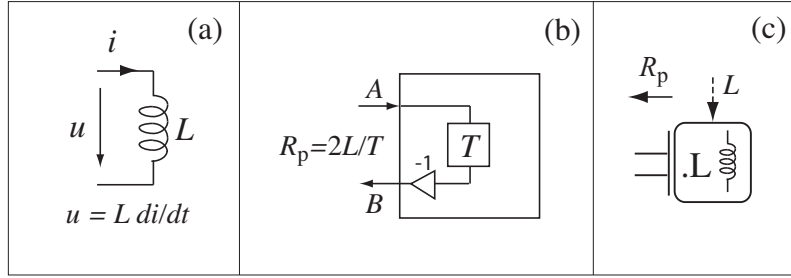


Figure 6.9: Inductance: (a) electrical inductor, (b) WDF inductor, and (c) symbol used for BlockCompiler.

By similar principles as for the capacitance above, an analog inductance can be approximated by a WDF element when selecting  $H(z) = -z^{-1}$  in Eq. (6.9) so that

$$R_p = 2L/T = 2 f_s L \quad (6.34)$$

$$H(z) = -z^{-1} \quad (6.35)$$

$$Z_L(z) = R_p \frac{1 - z^{-1}}{1 + z^{-1}} \quad (6.36)$$

At low frequencies Eq. (6.36) yields approximately

$$Z_L(j\omega) \approx \frac{j\omega R_p}{2f_s} \sim j\omega L \quad (6.37)$$

The behavior of this WDF inductance is compared to an analog inductance in Fig. 6.7, and the approximation error is plotted in Fig. 6.8.

Because for an inductor  $H(z) = -z^{-1}$  is an allpass function, the element is a lossless energy storage, see Eq. (6.4).

In the mechanical and acoustical domains the corresponding elements are mass (mechanical inductance) and acoustic inductance, respectively.

For BlockCompiler usage and implementation of inductances, see ??.

### 6.2.7 Generalized first order reflectance

The WDF resistance, capacitance, and inductance are special cases of first-order reflectance

$$H(z) = \frac{B(z)}{A(z)} = g z^{-1} \quad \rightarrow \quad Z(z) = R_p \frac{1 + g z^{-1}}{1 - g z^{-1}} \quad (6.38)$$

where  $g$  is given values in the range  $g = -1 \dots 1$  so that for inductance  $g = -1$ , for resistance  $g = 0$ , and for capacitance  $g = 1$ . For varying values of  $g$  the intermediate impedance values obtained are mixtures of reactive and resistive behavior as plotted in dotted lines in Fig. 6.7.

## 6.3 WDF elements with I-port

There exist useful versions of the previously described WDF elements that do not have a T-port, i.e., they have a wave port with a delay-free reflection (instantaneous response) from input  $A$  to output  $B$ . Therefore they have more stringent limitations where they can be applied for. They can be used only as a root element attached to the T-port of an adaptor (see next section). Such I-port elements are easily derived from the T-ones above by just selecting a port resistance different from the values used above.

### 6.3.1 Resistance with I-port

When the port resistance value of a resistor is forced to something different from  $R_p = R$ , let's say  $R_p = R_p^*$ , the resistance value  $R$  can be obtained with reflectance notation  $r = H$  from Eq. (6.9) by writing

$$Z = R = R_p^* \frac{1+r}{1-r} \quad (6.39)$$

which leads to

$$r = B/A = \frac{R - R_p^*}{R + R_p^*} \quad (6.40)$$

where  $r$  is a multiplier needed to implement the resistor as shown in Fig. 6.10.

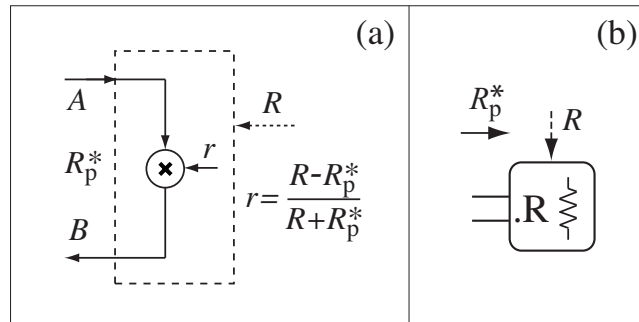


Figure 6.10: (a) Realization of I-resistor and (b) symbol used in BlockCompiler.

Short circuit and open circuit connections are special cases of impedance values. For a short circuit we get from Eq. (6.40)

$$Z \rightarrow 0 \quad \text{when} \quad r \rightarrow -1 \quad (6.41)$$

Notice that this does not depend on the (forced) port resistance. For the open circuit case we get, correspondingly,

$$Z \rightarrow \infty \quad \text{when} \quad r \rightarrow +1 \quad (6.42)$$

The open and short circuit connections have I-ports and therefore they can be used only as root elements of adaptor networks (Section 6.5.1).

### 6.3.2 Voltage and current sources with I-port

The behavior of a voltage source with a I-port that has (forced) port resistance  $R_p^*$  and internal resistance  $R_i$  can be derived from Eq. (6.15) as

$$B = \frac{R_p^*}{R_p^* + R_i} E - \frac{R_p^* - R_i}{R_p^* + R_i} A \quad (6.43)$$

where  $E$  is the open-circuit voltage at the port. For a I-port current source the open-circuit voltage is  $E = JR_i$ , where  $J$  is the short-circuit current, so that

$$B = \frac{R_p^* R_i}{R_p^* + R_i} J - \frac{R_p^* - R_i}{R_p^* + R_i} A \quad (6.44)$$

Figure 6.11 shows the symbols for I-port sources used in BlockCompiler (compare with Figs. 6.4 and 6.5).

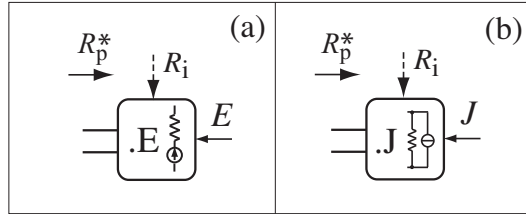


Figure 6.11: I-port source symbols used for BlockCompiler: (a) voltage source and (b) current source.

Ideal voltage and current sources are also useful WDF elements (with I-port limitations of usage). For an ideal voltage source, inserting  $R_i = 0$  in Eq. (6.43) yields

$$B = E - A \quad (6.45)$$

The behavior is independent of the (forced) port resistance. For an ideal current source  $R_i \rightarrow \infty$  so that from Eq. (6.44)

$$B = R_p^* J + A \quad (6.46)$$

As with real electric circuits, the user should avoid short circuit loading of a voltage source of zero or very low internal resistance. The same is true with high impedance loading of an ideal (or close to ideal) current source. In numeric simulation such overloading conditions result in numerical inaccuracy or over/underflow.

### 6.3.3 Example of simple nonlinearity: ideal diode

Nonlinearities and time-invariances require in most cases some special treatment in order to avoid problems of computability, stability, and aliasing problems. These questions are discussed more thoroughly in Chapter 9. Here we present a simple example, the ideal diode that has zero resistance for positive voltage over the diode and infinite resistance in the opposite direction.

The ideal diode can be realized as an element with a I-port by using the resistor in Subsection 6.3.1 and in Fig. 6.10, by making the resistance value dependent on the incoming wave variable. As can be concluded easily from Eq. (6.40), for positive values  $A > 0$ ,  $Z \rightarrow 0$ . Correspondingly



for negative values  $A < 0$ ,  $Z \rightarrow \infty$ . The diode is then computed by switching the value of  $r$  depending on the sign of  $a$  as

$$r = \begin{cases} -1 & \text{for } a \geq 0 \\ +1 & \text{for } a < 0 \end{cases} \quad (6.47)$$

This example gives a hint about the realization of arbitrary memoryless nonlinearities with a I-port, as will be presented in Section ??.

## 6.4 Adaptors

*Adaptors* [65, 58] are WDF elements that are used to create WDF structures by parallel and series connections of elements and substructures, i.e., to make circuits and networks as they are called in the electrical domain. Adaptors help to do this in a computable manner avoiding delay-free loops. Their operation is based on physical continuity laws, called the Kirchhoff laws in the electrical domain. The principles of N-port, 3-port, and 2-port adaptors are presented below in the form used in BlockCompiler.

### 6.4.1 Parallel N-port adaptor

The derivation of an  $N$ -port *parallel adaptor* resembles that of the DWG parallel junction in Section 4.1.4, particularly Eqs. (4.10)-(4.19) and Fig. 4.7. Figure 6.12 shows a parallel connection of  $N$  ports, where ports 1, 2, and  $N$  are explicitly drawn.

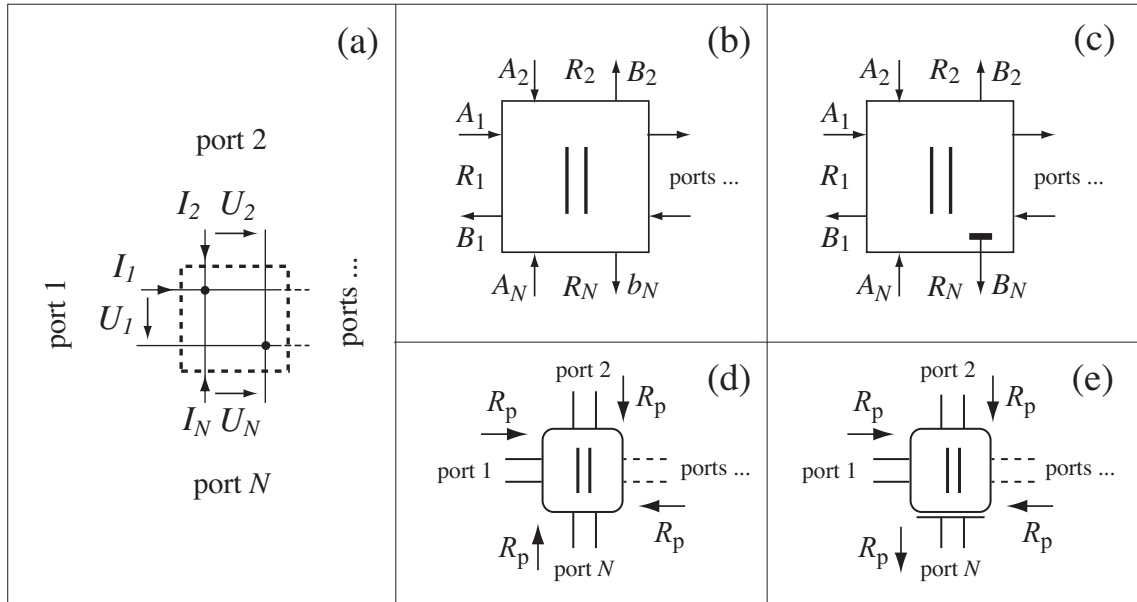


Figure 6.12: (a) Parallel connection of ports  $1 \dots N$ , (b) general WDF parallel adaptor, and (c) WDF parallel adaptor with a T-port (reflection-free port) drawn by a  $\top$  symbol. In (d) and (e) the symbols for BlockCompiler are for the general case and the T-port case, respectively, including the directions of port resistance data flow.

The Kirchhoff laws of continuity state that the voltages  $U_n$  at the ports  $n = 1 \dots N$  must be equal ( $U_{\text{par}}$ ) and the sum of currents  $I_n$  must be zero, i.e.,

$$U_n = U_{\text{par}} \quad (6.48)$$

$$\sum_{n=1}^N I_n = 0 \quad (6.49)$$

We can change to WDF wave variables using Eqs. (6.3) to write

$$\begin{cases} 2A_n = U_n + R_n I_n \\ 2B_n = U_n - R_n I_n \end{cases} \quad (6.50)$$

By eliminating K-variables  $U$  and  $I$  in Eqs. (6.48) and (6.49) we get

$$B_n = \left( \sum_{k=1}^N \gamma_k A_k \right) - A_n \quad (6.51)$$

$$\gamma_k = 2G_k / \sum_{i=1}^N G_i \quad (6.52)$$

where  $G_i = 1/R_i$  are port conductances. Coefficients  $\gamma_k$  correspond to the wave scattering coefficients  $\alpha_n$  for DWG junctions in Eq. (4.18). It can be readily seen that  $\sum_{k=1}^N \gamma_k = 2$ .

The total power flow to the ports of the parallel adaptor is zero, which follows immediately from Eqs. (6.48) and (6.49) by  $W = \sum U_n I_n = U_{\text{par}} \sum I_n = 0$  for any time moment. Thus the adaptor is lossless (energy-preserving).

From Eq. (6.51) it can be concluded that reflected waves  $B$  can in a general case be computed only when incoming wave  $A$  sample is available before  $B$  sample is needed, otherwise a delay-free loops appear. The WDF formulation allows for extra flexibility when one port is made free of immediate reflection. This is achieved by making the corresponding scattering coefficient  $\gamma$  equal to 1. For example for port  $N$  we can select  $G_N = \sum_{n=1}^{N-1} G_n$ , whereby  $\gamma_N = 1$  and for  $n = 1 \dots N$  we can write

$$\gamma_n = G_n / G_N \quad (6.53)$$

$$B_N = \sum_{n=1}^{N-1} \gamma_n A_n \quad (6.54)$$

$$B_n = B_N + A_N - A_n \quad (6.55)$$

The reflection-free port  $N$  is here called *T-port*<sup>3</sup> see Fig. 6.12(c) with symbol  $\top$ . Only one port of an adaptor can be made a T-port. For all others, called here I-ports,  $B_n$  is dependent on  $A_n$ .

### Example

The direction conventions shown in Fig. 6.12 specify the interpretation of positive voltages and currents in a circuit connected by a parallel adaptor. As a simple example, Fig. 6.13 depicts a circuit consisting of a voltage source (1.5 V, 1  $\Omega$ ) and two resistances (1  $\Omega$ ) in parallel.

<sup>3</sup>In literature such ports are typically called *reflection-free ports* [58] (or *adapted ports* [60]). This may be a bit confusing, because such ports in general are only free of instantaneous reflection; the term ‘port that is free of instantaneous reflection’ is inconvenient. Another fact is that such ports in our formulation are ‘controller’ ports that transmit port resistance (impedance) to an attached port of an adaptor or element with ‘controlled’ port resistance. Thus such ‘transmitter’ ports are called here *T-ports* and the ‘receiver’ ports are called *I-ports*.

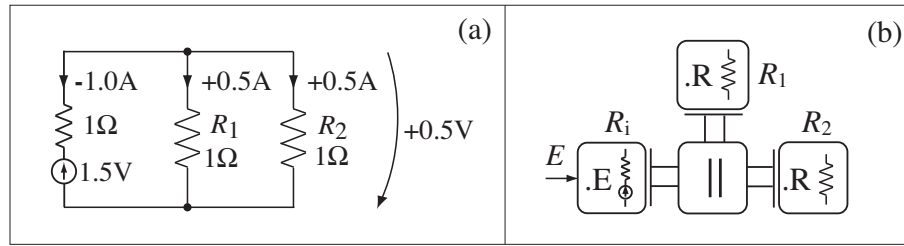


Figure 6.13: Simple parallel connection of a voltage source and two resistances.

Due to loading the source by  $0.5 \Omega$  resistance (two  $1 \Omega$  resistors in parallel), the voltage across all elements is  $+0.5 \text{ V}$ , and the current through each load resistance is  $+0.5 \text{ A}$ . The current in the voltage source element, however, flows opposite to the direction convention, therefore being  $-1 \text{ A}$ .

### 6.4.2 Series N-port adaptor

The derivation of a WDF  $N$ -port *series adaptor* can be done in a way analogous to the WDF parallel adaptor above. Figure 6.14 shows a series connection of  $N$  ports, where ports 1, 2, and  $N$  are explicitly drawn.

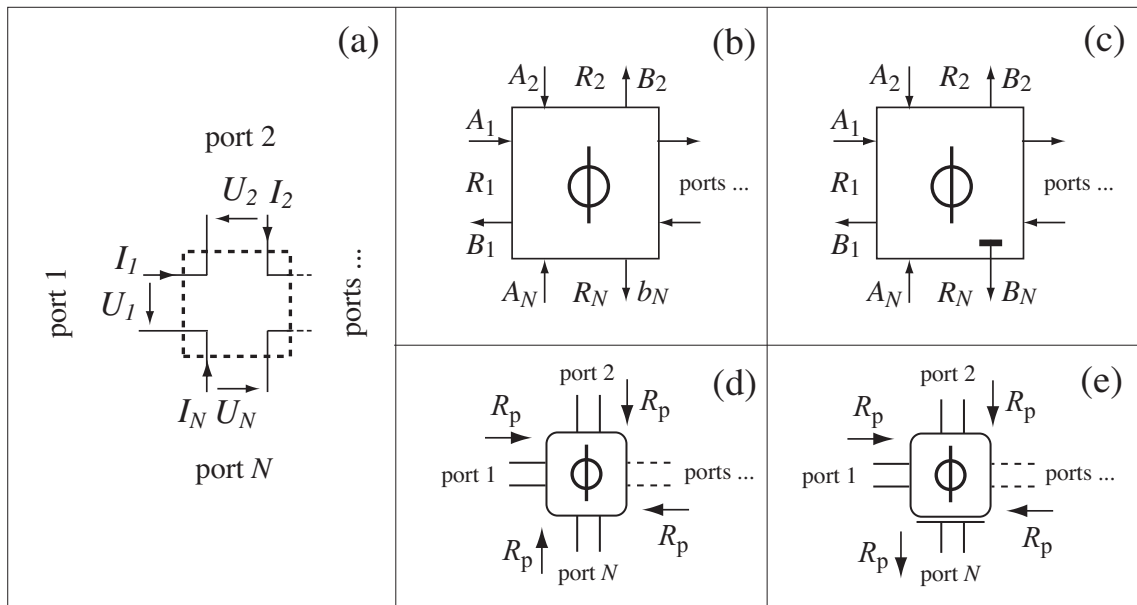


Figure 6.14: (a) Series connection of ports  $1 \dots N$ , (b) general WDF series adaptor, and (c) WDF series adaptor with a T-port (reflection-free port) drawn by a  $\top$  symbol. In (d) and (e) the symbols for BlockCompiler are for the general case and the T-port case, respectively.

The Kirchhoff laws for a series connection require that the currents  $I_n$  at the ports must be equal ( $I_{\text{ser}}$ ) and the sum of voltages  $U_n$  around the series connection must be zero, i.e.,

$$I_n = I_{\text{ser}} \quad (6.56)$$

$$\sum_{n=1}^N U_n = 0 \quad (6.57)$$

From Eqs. (6.3), (6.56) and (6.57) and by eliminating K-variables we can write

$$B_k = A_k - \gamma_k \sum_{n=1}^N A_n \quad (6.58)$$

$$\gamma_k = 2R_k / \sum_{n=1}^N R_n \quad (6.59)$$

where  $R_n$  are port resistances. Coefficients  $\gamma_k$  are wave scattering coefficients in a series connection, for which  $\sum_{k=1}^N \gamma_k = 2$ .

As for the parallel adaptor, the total power flowing to the ports of the series adaptor is zero, i.e., the adaptor is lossless (energy-preserving).

As for the parallel adaptor, a T-port is achieved by making the corresponding scattering coefficient  $\gamma$  equal to 1. For example for port  $N$  (see Fig. 6.14(c)) we can select  $R_N = \sum_{n=1}^{N-1} R_n$ , whereby  $\gamma_N = 1$  and for  $n = 1 \dots N$  we can write

$$\gamma_n = R_n / R_N \quad (6.60)$$

$$B_N = -\gamma_N \sum_{n=1}^{N-1} A_n \quad (6.61)$$

$$B_n = A_n - \gamma_n (A_N - B_N) \quad (6.62)$$

Only one port of a series adaptor can be made a T-port.

### Example

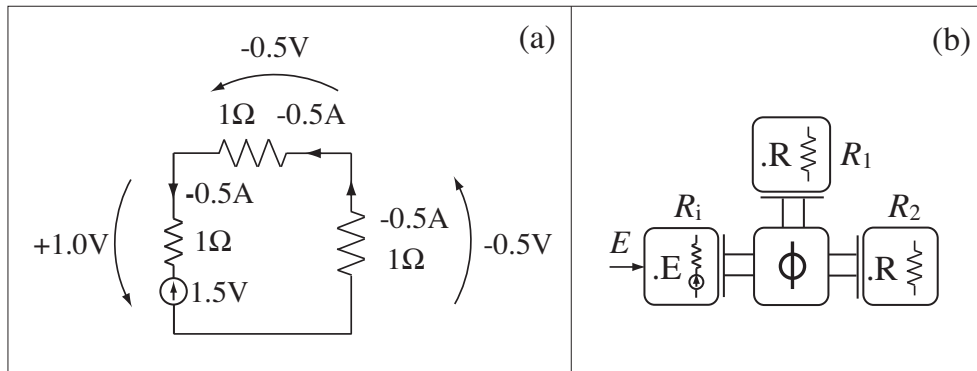


Figure 6.15: Simple series connection of a voltage source and two resistances.

Figure 6.14 specifies the positive voltage and current directions in a circuit connected by a series adaptor. As an example, Fig. 6.15 depicts a circuit consisting of a voltage source (1.5 V, 1 Ω) and two resistances (1 Ω) in series.

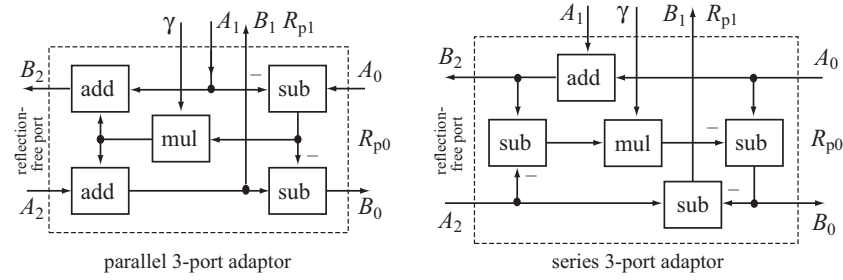


Figure 6.16: Optimized realization of 3-port adaptors: parallel (left) and series (right) adaptor.

Now all the port currents are equal to  $-0.5$  A, i.e., negative, because the current flows clockwise, while the positive direction is anticlockwise. The voltage of the source port is  $+1$  V, while for the two resistances the port voltage is  $-0.5$  V. This can be a bit surprising unless thinking carefully the positive directions specified. Notice the difference to the parallel adaptor before, where the voltages across all elements were positive.

### 6.4.3 3-port adaptors

The algorithm of computing a parallel or series adaptor can be optimized to use a minimum number of operations for a given number of ports. Figure 6.16 describes the realization of such parallel and series 3-port adaptors, consisting of adders, subtractors, and multipliers. The parameter  $\gamma$  is computed as

$$\gamma = R_{p,0}/(R_{p,0} + R_{p,1}) \quad (6.63)$$

3-port adaptors are useful in implementing binary tree structures, which are universal in the sense that multiport adaptors can be realized using them.

### 6.4.4 Parallel 2-port adaptor

Two-port adaptors are of special interest because often there is need to connect together two T-ports that have different port impedances. The two-port parallel adaptor is a WDF equivalent of the Kelly-Lochbaum junction (Section ??). In a similar way the scattering equations can be derived to be

$$\begin{cases} B_1 = \gamma A_1 + (1 - \gamma) A_2 \\ B_2 = (1 + \gamma) A_1 - \gamma A_2 \end{cases} \quad (6.64)$$

where  $\gamma$  is the reflection coefficient obtained from port resistance as

$$\gamma = \frac{R_{p2} - R_{p1}}{R_{p2} + R_{p1}} \quad (6.65)$$

For a computationally more efficient one-multiplier formulation Eq. (6.66) can now be written

$$\begin{cases} B_1 = A_2 + \gamma(A_1 - A_2) \\ B_2 = A_1 + \gamma(A_1 - A_2) \end{cases} \quad (6.66)$$

which is shown as a realization diagram in Fig. 6.17.

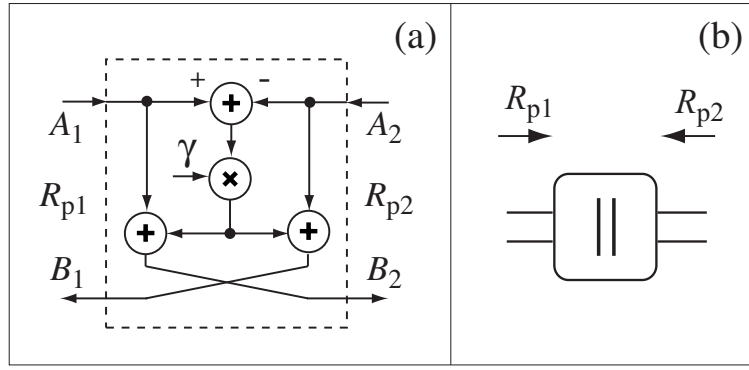


Figure 6.17: (a) Efficient realization of a two-port parallel adaptor and (b) symbol used for BlockCompiler.

### 6.4.5 Series 2-port adaptor

The series two-port adaptor is similar to the parallel two-port adaptor, except that the voltage and current signs at the second port are negated (see series adaptor discussion above). Therefore the need for a separate series 2-port adaptor is not very frequent, and the parallel 2-port adaptor is recommended in most cases to be used instead.

### 6.4.6 WDF adaptors vs. DWG scattering junctions

A comparison of the DWG parallel and series scattering junctions in Section 4.1.4 and the WDF adaptors above makes it clear that these are just two different formulations of wave scattering. Therefore they can be seen rather as somewhat different computational realizations of the same physical phenomenon. Furthermore, by using the WDF wave definitions in Eq. 6.3 the DWG and WDF wave variables as well as the corresponding ports have been made directly compatible.

This equivalence gives freedom to construct discrete-time physical models on either DWG or WDF formulations or their combinations. In fact, it is somewhat artificial to make clear distinction between them, because they are just different approaches to W-modeling, DWGs approaching from a distributed element and WDFs from a lumped element points of view. It is more a practical choice how to implement a W-model.

A practical convenience achieved by using WDF adaptors for port interconnectivity is that the computational impedances (port resistances) are always real-valued and no digital filters are needed inside adaptors, while for frequency-dependent impedances the DWG junctions become multiple-input-multiple-output digital filters. Therefore in the practical implementation of W-modeling in BlockCompiler we prefer the WDF formulations.

Another advantage of WDF adaptors presented here over the DWG junctions discussed in Section 4.1.4 is that the T-(reflection-free) ports allow for connecting together subsystems in series or parallel, while the junctions can be connected only through delayed elements in order to avoid delay-free loops<sup>4</sup>.

<sup>4</sup>In fact, DWG junctions can in principle be formulated to include “T-ports” for further interconnections by creating Thevenin and Norton equivalent ports, but for frequency-dependent impedances this leads to complicated computation. Also for WDFs, it is possible to use generalized (frequency-dependent) port impedances instead of port resistances, but normally this only brings excessive complexity to computation.

## 6.5 Network structures and interconnection rules

WDF adaptors are used to interconnect elements through their wave ports into parallel and series networks. There are, however, some restrictions on the connectivity of different port types and therefore also on possible network structures.

### 6.5.1 Tree structures

An  $n$ -port adaptor (parallel or series) described above allows for connecting any number of elements with T-ports into a star-like structure (see Figs. 6.12(b) and 6.14(b)). Such a parallel or series network can be connected to a further structure through one but only one port, because only one T-port is available for any adaptor (see Fig. 6.12(c) and 6.14(c)). This means that only *tree structures*, not more general graphs with loops, can be built, unless there are delays (delay-lines) with two T-ports to ‘isolate’ the substructures. Figure 6.18 presents an example of a tree structure made using two  $n$ -port adaptors. In a network there may or may not be a single root element with a I-port (dashed line  $E_6$  in this case).

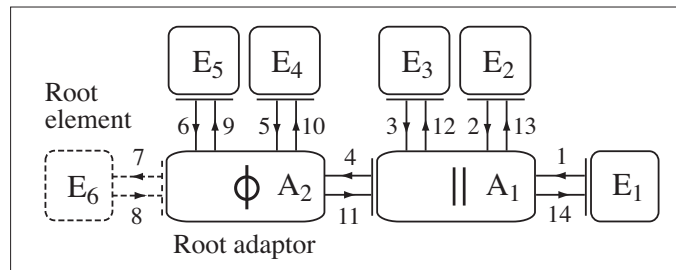


Figure 6.18: A WDF network made of basic elements  $E_1 \dots E_6$  and adaptors  $A_1$  and  $A_2$ . Numbers 1...14 describe the data flow order of computing operations. Optional root element  $E_6$  is marked with dashed line. If there is no root element,  $A_2$  will be the root adaptor of the network.

Figure 6.18 shows also one possible execution order (schedule) of operations that is computable. For the T-ports of basic elements the reflected wave value  $B$  can be read immediately without need to know the incident wave  $A$ , while for the T-port of a parallel or series adaptor the  $B$ -value is obtained by combining the  $A$ -values of the other ports as shown in Figs. 6.12(b) and 6.14(b). Thus the computation starts from the leaves of the tree and progresses to the root element of the tree. Thereafter the computation proceeds back towards the leaves until the whole network has been traversed for one sample step in time [61]. At the next step the same is repeated again.

From this scheduling principle it follows that there are three valid cases of port connections:

1. The basic form of connection is between a T-ports and a I-port (of the same physical domain), where the T-port propagates the port resistance value to the I-port.
2. The two-port adaptors in Sections 6.4.4 and 6.4.5 connect to two T-ports (of the same physical domain) by realizing wave scattering between different port resistances.
3. Two T-ports with the same port resistance (and same physical domain) can be connected directly without an adaptor, because no scattering of waves takes place.

### 6.5.2 Binary trees

Any tree structure can be represented also as a computationally equivalent *binary tree*. This means that an arbitrary WDF structure can be composed as a binary tree by using 3-port adaptors only [61]. Therefore using n-port adaptors is not a necessity<sup>5</sup>. Figure 6.19 shows how an n-port adaptor is converted into a binary tree.

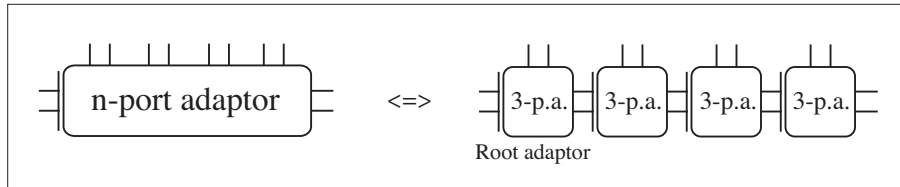


Figure 6.19: N-port adaptor (left) vs. equivalent binary tree of 3-port adaptors (right).

### 6.5.3 Root elements of a WDF tree structure

The block named  $E_6$  and drawn in dashed line in Fig. 6.18 is of special importance in working with WDFs. As discussed before, a WDF network (without delay elements) can contain one and only one element with I-port to avoid delay-free loops. Root elements with a I-port have special freedom, and therefore they can be used to realize for example:

- In the root position an element with a I-port is independent in the sense that changes in it do not have any effect to the port impedances of the port impedances in other parts of the network.
- Particularly the root element can be nonlinear, time-varying, or parametrically controlled without global parametric effects. For example nonlinear resistors can be used.
- Also nonlinear reactances, such as nonlinear capacitors and inductors realized through mutator, discussed in Section 6.6.6 below, can be used as root elements.

A problem in designing for example nonlinear root elements is that the element characteristics are typically given in in K-variables, such as current as a function of voltage  $i = f(u)$ , while the port variables are W-variables  $a$  and  $b$ . It turns out that mapping between them in most cases is quite inconvenient due to implicit equations. One possibility is to use iterative solution. If this is too time-consuming at runtime for real-time applications, a more efficient solution is to compute off-line a look-up-table, which is at runtime used with interpolation. These techniques are studied in Chapter 9.

### 6.5.4 Initialization of WDF networks

Initialization of a network consisting of WDF elements and adaptors means setting proper initial values for the wave variables. It requires special consideration because the global Kirchhoff constraints must be fulfilled at once for the initial state.

<sup>5</sup>In BlockCompiler, adaptors are in most cases created implicitly when calling functions that make parallel (`.par`) or series (`.ser`) connections, and the system takes care whether n-port, 3-port, 2-port, or direct port connections are created, see Section ?? . Thus the user does not necessarily deal explicitly with adaptors. In BCT (Binary Connection Tree) [61], WDF structures are built of binary trees.



Problems appear particularly with the reactances (capacitors and inductors) and possible nonlinearities. At the initial simulation moment  $t = 0$  each loaded capacitor acts as an ideal voltage source of the initial voltage (or related charge) and infinite port resistance. Each inductor acts correspondingly as an ideal current source (infinite port resistance). Dealing with multiple ideal sources (and nonlinearities) does not easily fit to regular WDF simulation. Therefore another strategy is to be taken.

The initial state of a network can be solved separately by using the Thévenin or Norton equivalents (Section ??) of the network<sup>6</sup> [?, 60, ?]. This follows a similar two-pass schedule as in the regular WDF tree traversing described in Section 6.5.1. The difference is that K-variables (voltage and current in the electrical domain) are used instead of W-variables. Each element is represented as a Thévenin (or Norton) equivalent, taking capacitors and inductors as voltage and current sources, respectively. This is continued to the T-ports of adaptors up to the root adaptor and possible root element, which needs special treatment if it is a nonlinear one. Thereafter the computation passes backwards to the leafs of the tree, by computing first the K-variables and then the W-variables using Eq. (6.3), until all ports get their initial W-variable values.

If there is a nonlinear root element, its K-variables must be solved separately. Figure 6.20 plots the principle of finding the voltage and current when the Thévenin equivalent of a network is described by the loading line and a nonlinear element is given by its nonlinear  $U$ - $I$  characteristic curve. In practice this may require iteration or some other indirect solving method. Initialization will be even more complicated if there are multiple nonlinearities or controlled sources.

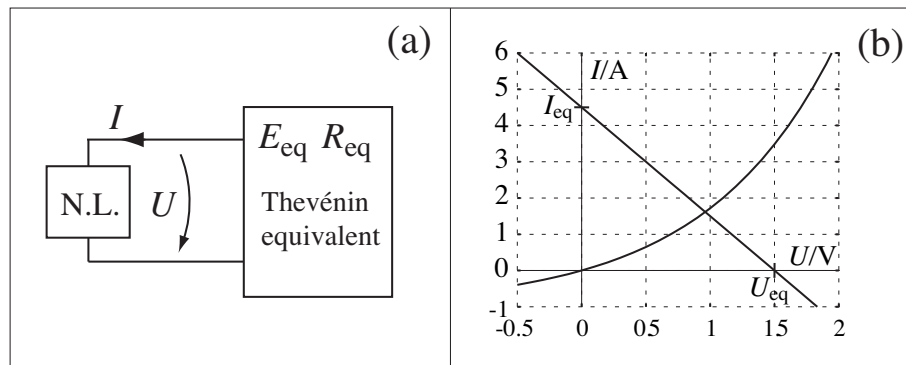


Figure 6.20: (a) Nonlinear root element (N.L.) connected to Thévenin equivalent of a WDF network and (b) graphical presentation of solving voltage and current. The Thévenin equivalent has  $E_{eq} = 1.5$  V and  $R_{eq} = 1/3$   $\Omega$ . The nonlinear resistor has characteristics  $I = 3 \exp(U) - 1$ . Solution is represented by the crossing point of the curves.

## 6.6 WDF 2-port and N-port elements

There are a few important two-port WDF elements that help in building physics-based models in one or multiple physical domains. The transformer are needed to change impedance levels, the gyrator maps capacitances to inductances and vice versa, and the mutator maps a resistance to a reactance and vice versa. Based on these basic elements (and some other types not described

<sup>6</sup>In the current version of BlockCompiler a simpler approximate method of initialization is used,

here) it is possible to construct new two- and multiport elements, such as transducers between different physical domains.

### 6.6.1 Ideal transformer

An ideal transformer is a two-port lossless element that maps the impedance level at one port to another level at the other port, see Fig. 6.21.

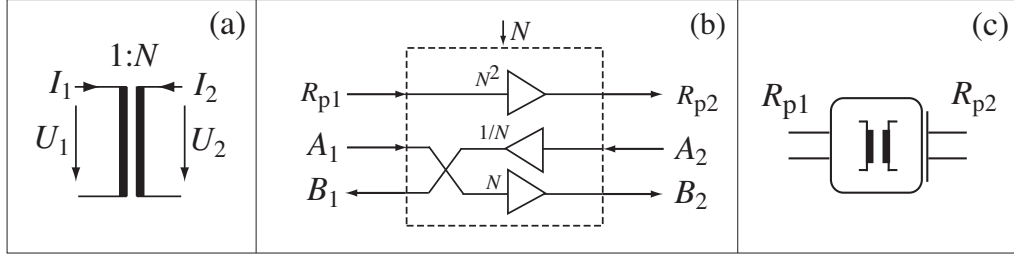


Figure 6.21: Ideal transformer as a two-port element: (a) electrical symbol and (b) WDF principle and (c) symbol for BlockCompiler.

When using K-variables voltage ( $U$ ) and current ( $I$ ) for the two ports, the ideal transformer does the transform:

$$\begin{cases} U_2 = N U_1 \\ I_2 = (1/N) I_1 \end{cases} \quad (6.67)$$

where  $N$  is the transform factor (“turns ratio”), which can be positive or negative. Thus the power transform is  $P_2 = U_2 I_2 = N U_1 (1/N) I_1 = P_1$ , which means energy preservation. Impedances are transformed by

$$Z_2(z) = U_2(z)/I_2(z) = \frac{N U_1(z)}{(1/N) I_1(z)} = N^2 Z_1(z) \quad (6.68)$$

This can be obtained by port impedance mapping

$$R_{p2} = N^2 R_{p1} \quad (6.69)$$

Voltages and currents are scaled properly when the wave variables  $a$  and  $b$  are scaled by

$$B_2 = N A_1 \quad \text{and} \quad B_1 = (1/N) A_2 \quad (6.70)$$

(see Fig. 6.21), which means that

$$\begin{cases} U_2 = A_2 + B_2 = N A_1 + N B_1 = N U_1 & \text{and} \\ I_2 = (A_2 - B_2)/R_{p2} = (N A_1 - N B_1)/(N^2 R_{p1}) = (1/N) I_1 \end{cases} \quad (6.71)$$

as was targeted in Eq. (6.67).

The usage of WDF ideal transformers in BlockCompiler is presented in Section ??.

### 6.6.2 Gyrator

*Gyrator* is a particularly simple two-port element, shown in Fig. 6.22, that maps a capacitor to an inductor and vice versa. More generally,

$$U_2 = A_2 + B_2 = \frac{A_1 - B_1}{R_p} R_p = I_1 R_p \quad (6.72)$$

$$I_2 = \frac{A_2 - B_2}{R_p} = \frac{-B_1 - A_1}{R_p} = -U_1 / R_p \quad (6.73)$$

$$Z_2 = \frac{U_2}{-I_2} = \frac{R_p^2}{Z_1} \quad (6.74)$$

A resistor is mapped to a resistor of the same value, because for it  $Z_2 = R_p = R$ .

The usage of WDF gyrators in BlockCompiler is described in Section ??.

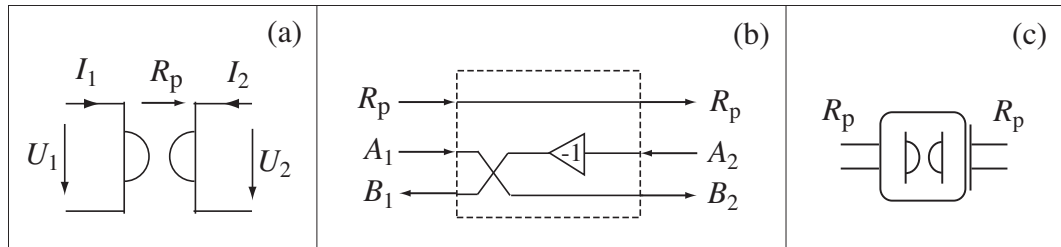


Figure 6.22: Gyrator as a two-port element: (a) electrical symbol and (b) WDF principle and (c) symbol used for BlockCompiler.

### 6.6.3 Dualizer

As discussed in ??, there exists duality between models so that exchange of impedances  $\leftrightarrow$  admittances, potential  $\leftrightarrow$  flow variables (e.g., voltage  $\leftrightarrow$  current), and series  $\leftrightarrow$  parallel connections yields a model with an equivalent behavior. This duality is a useful principle, particularly when modeling a physical system by an inverse (indirect) analogy in another domain, or when modeling transducers between domains. By looking at Eq. (6.74) it becomes obvious that a *dualizer*, i.e., duality mapping is obtained by cascading a gyrator of Fig. 6.22 and an ideal transformer of Fig. 6.21 with a turns ratio of  $R_p : 1$  so that

$$U_2 = I_1 \quad (6.75)$$

$$I_2 = -U_1 \quad (6.76)$$

$$Z_2 = 1/Z_1 = Y_1 \quad (6.77)$$

### 6.6.4 Circulator

*Circulator* is an element that circulates the wave variables around its (three or more) ports as shown in Fig. 6.24. It is the WDF equivalent of circulators used in microwave technology.

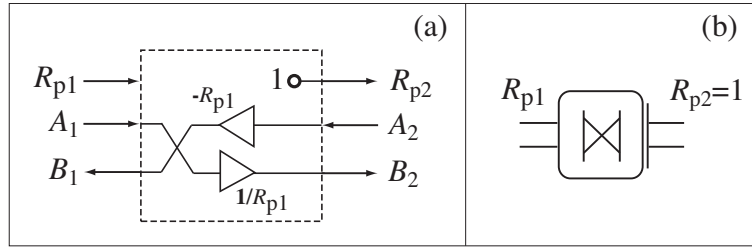


Figure 6.23: Dualizer as a two-port element: (a) WDF principle and (b) symbol used for Block-Compiler.

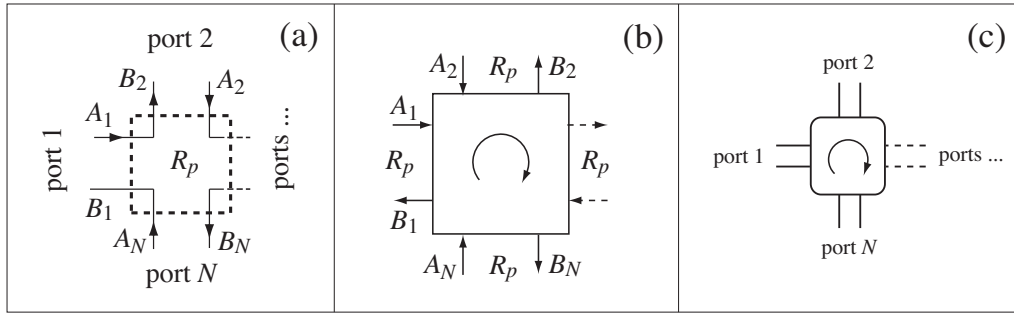


Figure 6.24: (a) Circulator principle, (b) WDF symbol, and (c) symbol used for BlockCompiler.

### 6.6.5 Unit element

The *unit element* is the building block in the original WDF theory that comes close to the digital waveguide idea in the sense of simulating wave propagation. Figure 6.25 shows the formulation of a unit element with half sample delays between ports ( $\Delta = 0$ ). This requires double sample rate to be used.

A variant of the unit element where delay is distributed asymmetrically between the two directions is called QUARL (quasi-reciprocal line), where  $\Delta \neq 0$ . The extreme cases are when  $\Delta = T/2$  or  $\Delta = -T/2$ , so that a single unit delay is only in one direction.

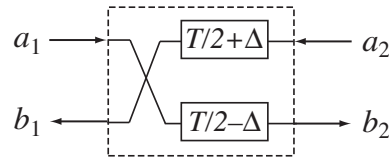


Figure 6.25: Unit element ( $\Delta = 0$ ) and QUARL ( $\Delta \neq 0$ ).

### 6.6.6 Mutators and resistance-reactance transforms

The *WDF mutator* [60] is a lossless two-port element that transforms a resistance to a reactance, i.e., to a capacitance or an inductance. The basic idea is to modify the 2-port adaptor in Fig. 6.17 by replacing the reflection coefficient  $\gamma$  by a memory element. Figure 6.26 depicts the case where  $\gamma = z^{-1}$ , a unit delay.

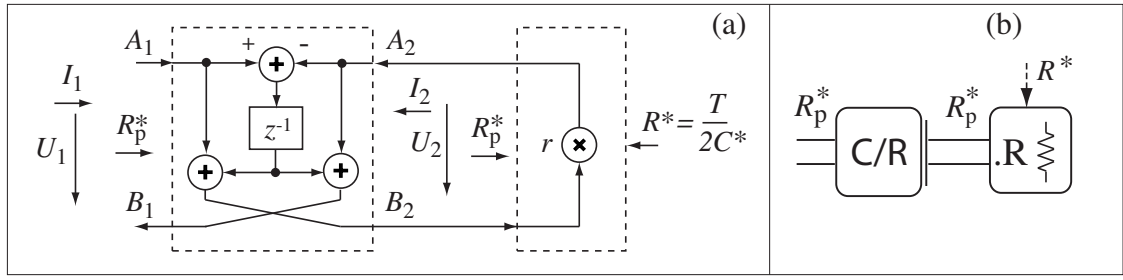


Figure 6.26: (a) Mutator (left-hand block) that transforms a (delay-free) resistor (right-hand block) into a capacitor behavior seen at the left-hand mutator port. (b) Symbol for the capacitive mutator.

### Realization of capacitance through a mutator

When the right-hand port of such a modified adaptor is connected to a resistor with a I-port introduced in Section 6.3.1, the behavior of the system in Fig. 6.26 can be described by equations

$$B_1(z) = z^{-1}A_1(z) + (1 - z^{-1})A_2(z) \quad (6.78)$$

$$B_2(z) = (1 + z^{-1})A_1(z) - z^{-1}A_2(z) \quad (6.79)$$

$$A_2(z) = rB_2(z) \quad (6.80)$$

where  $r$  is a coefficient to control the resistance value of the right-hand block as specified by Eq. (6.40). It is essential to recognize that the port resistance  $R_p^*$  is obtained from the circuit to which the mutator's left-hand port is attached. The port behavior can be solved as

$$H(z) = \frac{B_1(z)}{A_1(z)} = \frac{r + z^{-1}}{1 + rz^{-1}} \quad (6.81)$$

i.e., the reflectance  $H(z)$  is a first-order allpass filter. This means that it corresponds to a lossless element. We can easily see that when  $r = 0$  then  $H(z) = z^{-1}$ , which corresponds to a capacitor according to Eq. (6.25). More generally, the impedance seen at the left-hand port of the mutator is

$$Z(z) = R_p^* \frac{1 + H(z)}{1 - H(z)} = R_p^* \frac{1 + r}{1 - r} \cdot \frac{1 + z^{-1}}{1 - z^{-1}} \quad (6.82)$$

When setting the impedances of Eqs. (6.23) and (6.82) equal with notations  $C = C^*$  and  $R_p = R_p^*$ , we can notice that the mutated resistor appears as a WDF capacitance of value

$$C^* = \frac{T}{2R_p^*} \cdot \frac{1 - r}{1 + r} \quad (6.83)$$

where  $T$  is the sample interval. By varying  $r$  from -1 through 0 to 1 the capacitance value varies from 0 through  $T/(2R_p^*)$  to  $\infty$ . For a desired capacitance value  $C^*$  the control parameter  $r$  can be obtained as

$$r = \frac{T - 2R_p^*C^*}{T + 2R_p^*C^*} = \frac{1 - 2f_s R_p^* C^*}{1 + 2f_s R_p^* C^*} \quad (6.84)$$

This corresponds to a resistor with a I-port (right-hand block in Fig. 6.26), resistance value  $R^*$  being set to

$$R^* = \frac{T}{2C^*} \quad (6.85)$$

The capacitance realized by the mutator raises the question of what is obtained this way, because the realization is certainly more complex than the capacitance derived in Section 6.2.5. Furthermore, usage of the mutated capacitance is restricted to the single root element of a circuit only. The advantage gained in this way is that the mutated capacitance can be time-varying or nonlinear (see Chapter 9 for further discussion) without global effects to port resistances.

The role of the mutator can be explored further by checking how it transforms the K-variables at its ports. Referring to Fig. 6.26 the ratio of voltages  $U_R (= U_2)$  and  $U_1$  can be solved using Eqs. (6.78)-(6.80) as

$$\frac{U_R}{U_1} = 1 \quad (6.86)$$

and the ratio of currents  $I_R (= -I_2)$  and  $I_1$  becomes

$$\frac{I_R}{I_1} = \frac{1 + z^{-1}}{1 - z^{-1}} \quad (6.87)$$

The ratio of currents corresponds to a (bilinear) integrator, which means that the current at the resistor port is proportional to the charge of the mutated capacitor (or displacement as the integral of velocity in the mechanical domain).

### Realization of inductance through a mutator

When the unit delay ( $\gamma = z^{-1}$ ) is replaced by  $\gamma = -z^{-1}$  in the mutator of Fig. 6.26, the resulting element will be an inductor

$$L^* = \frac{TR_p^*}{2} \cdot \frac{1 + r}{1 - r} \quad (6.88)$$

for which varying  $r$  from -1 through 0 to 1 the inductance value varies from 0 through  $TR_p^*/2$  to  $\infty$ . This corresponds to a resistor with a I-port (right-hand block in Fig. 6.26), resistance value  $R^*$  being set to

$$R^* = \frac{2L^*}{T} \quad (6.89)$$

The inductance obtained this way has the same advantages and limitations as the mutated capacitor above. The relations of K-variables of the mutator ports, with notations used in Fig. 6.26, are

$$\frac{U_R}{U_1} = 1 \quad (6.90)$$

$$\frac{I_R}{I_1} = \frac{1 - z^{-1}}{1 + z^{-1}} \quad (6.91)$$

which means that the current at the resistor is a (bilinearly) differentiated version of the current at the mutated inductor port.

## 6.7 Transducers as inter-domain two-port elements

Transducers are physical devices that make coupling and energy/signal transfer between subsystems in different physical domains. Typical examples thereof are the electrodynamic used in loudspeakers and microphones. In a loudspeaker, electrical vs. mechanical transduction and

then the mechanical vs. acoustical transduction takes place so that a full model includes an electrical  $\rightarrow$  mechanical  $\rightarrow$  acoustical chain of coupled subsystems. As an example of transducers, in this subsection we describe the basic electrodynamic transducer. More realistic cases related to loudspeakers and microphones as well as other transducers are modeled in Section ??.

### 6.7.1 Example: Electrodynamic transduction

*Electrodynamic transducers* in their basic form are devices that realize transduction between the electrical and the mechanical domains. As discussed in Section ??, the basic electrodynamic transducer consists of a conducting wire in a magnetic field. If the wire has length  $l$  in a homogeneous field of flux density  $B$ , the connections between the domain variables are

$$\begin{cases} F_m = Bl I_e \\ U_e = Bl V_m \end{cases} \quad (6.92)$$

where  $F_m$  is the mechanical force acting to the wire due to the electric current  $I_e$  flowing in the wire, and  $U_e$  is the voltage induced across the wire due to the mechanical velocity  $V_m$  of the wire.

#### Mechanical mobility analogy

The electrodynamic transducer is often modeled so that the mechanical side is represented by a mobility (admittance) analogy. The following conventions are then applied to a generic WDF two-port element:  $F_m \rightarrow I_2$ ,  $V_m \rightarrow U_2$ ,  $U_e \rightarrow U_1$ ,  $I_e \rightarrow I_1$ , and  $Bl \rightarrow 1/N$ . The generic two-port equations become

$$\begin{cases} U_2 = NU_1 \\ I_2 = (1/N)I_1 \end{cases} \quad (6.93)$$

Comparison to Eq. (6.67) shows that the transducer can be realized by an ideal transformer with a turns ratio of  $1/Bl$ . For wave variables the two-port equations now become

$$B_2 = (1/Bl)A_1 \quad (6.94)$$

$$B_1 = BlA_2 \quad (6.95)$$

$$R_{p2} = (1/Bl)^2 R_{p1} \quad (6.96)$$

The WDF realization is shown in Fig. 6.27.

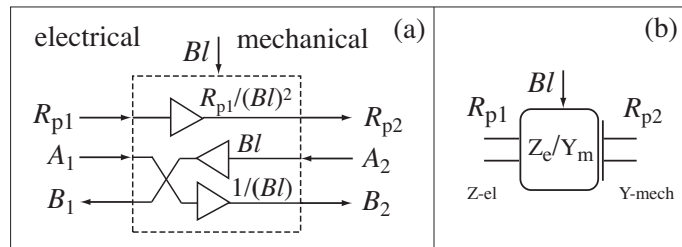


Figure 6.27: Electrodynamic transducer with mobility analogy on the mechanical side: (a) WDF implementation principle and (b) symbol used for BlockCompiler. Direction from electrical to mechanical domain for port resistance control is applied.

### Mechanical impedance analogy

When the following conventions are written to Eq. (6.92):  $F_m \rightarrow U_2$ ,  $V_m \rightarrow I_2$ ,  $U_e \rightarrow U_1$ ,  $I_e \rightarrow I_1$ , and  $Bl \rightarrow 1/N$ , then from Eq. (6.92) the generic two-port equations for K-variables become

$$\begin{cases} U_2 = (1/N)U_1 \\ I_2 = NU_1 \end{cases} \quad (6.97)$$

This shows that a transducer model with impedance analogy on the mechanical side can be realized by cascading an ideal transformer (Section 6.6.1) having a turns ratio of  $N = 1/Bl$  and a dualizer (Section 6.6.3). For wave variables the two-port equations become

$$B_2 = (Bl/R_{p1})A_1 \quad (6.98)$$

$$B_1 = (-R_{p1}/Bl)A_2 \quad (6.99)$$

$$R_{p2} = (Bl)^2 \quad (6.100)$$

Such a WDF two-port realization is presented in Fig. 6.28.

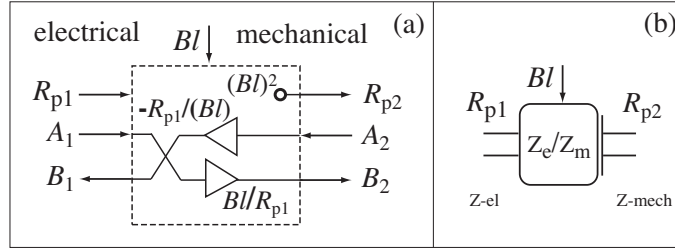


Figure 6.28: Electrodynamic transducer with impedance analogy on the mechanical side: (a) WDF implementation principle and (b) symbol used for BlockCompiler. Direction from electrical to mechanical domain for port resistance control is applied.

Similarly to other two-port elements, the electrodynamic transducer two-port in Fig. 6.28 is two-directional in signal flow, but the port impedance control flow is from the electrical to the mechanical domain. A transducer model with the opposite direction of control flow is easy to derive.

## 6.8 K-W conversion in lumped element models

The basic WDF elements, together with delay lines from the DWG approach, allow for constructing advanced circuit and network structures. Often it is beneficial to use not only compositions of the basic elements but also more complex entities that simulate a given impedance. Such a K-variable formulation may appear from analytical derivation of a physical system, such as the radiation impedance of a piston in Section ??, or from measurements of a real physical system such as in Section ?. Another reason to such entities is that even for circuits composed easily from basic elements it may be computationally more efficient to consolidate them into more compact digital filter forms for real-time simulation.

The digital waveguide scattering junctions in Section 4.1.4, see particularly Figs. 4.7 and 4.8, make it possible to use arbitrary impedances/admittances given in polynomial (FIR-like) or rational (IIR-like) form. In this section we discuss the means of realizing and approximating



arbitrary LTI impedances by FIR and IIR filtering techniques using WDF-compatible wave ports.

It is important to remember that the impedances given in polynomial or rational  $z$ -transform expressions are not filters, although they look like FIR and IIR filters. This is because an impedance is just a constraint between K-variables (e.g., voltage and current), not a causal input-output relationship. It must be converted to a port reflectance to make it a digital filter. As for many WDF elements above, by selecting a proper value of port resistance the reflectance can be made computable as a reflection-free T-port.

### 6.8.1 Impedance given in polynomial form

Let us first consider the case where an impedance is given as a polynomial form  $z$ -transform expression

$$Z(z) = \frac{U(z)}{I(z)} = \sum_{i=0}^{N-1} b_i z^{-i} \quad (6.101)$$

For compatibility to WDF models this K-variable expression must be converted to port reflectance  $H(z)$  with port resistance  $R_p$  by writing

$$H(z) = \frac{Z(z) - R_p}{Z(z) + R_p} = \frac{\sum_{i=0}^{N-1} b_i z^{-i} - R_p}{\sum_{i=0}^{N-1} b_i z^{-i} + R_p} \quad (6.102)$$

It can be used as such for any  $R_p$  within  $0 < R_p < \infty$ , if an I-port with delay-free reflection is allowed. In most cases we are however interested in realizing an element with a T-port so that it can be used freely in WDF models. Then the requirement is that reflection at time index 0 must be zero, which is fulfilled when

$$R_p = b(0) \quad (6.103)$$

$$H(z) = \frac{\sum_{i=1}^{N-1} b_i z^{-i}}{2R_p + \sum_{i=1}^{N-1} b_i z^{-i}} = \frac{\frac{1}{2R_p} \sum_{i=1}^{N-1} b_i z^{-i}}{1 + \frac{1}{2R_p} \sum_{i=1}^{N-1} b_i z^{-i}} \quad (6.104)$$

The reflectance  $H(z)$  will therefore be an IIR filter of the same order than the impedance polynomial. Notice also that  $b(0)$  must be non-zero, otherwise the port resistance becomes zero.

This mapping from  $Z(z)$  to  $H(z)$  is easy, but it would be nice to use the FIR type formulation of  $Z(z)$  more directly as a basis for the reflectance as well. The last form of (6.104) gives a hint about a realization that is close to the FIR filter structure. Figure 6.29 shows how the reflectance  $H(z)$  can be computed efficiently using the backbone of the regular FIR filter, just slightly modified to make an IIR filter. There is only little extra computation required to the regular FIR filter form<sup>7</sup>.

<sup>7</sup> Furthermore, in an object-oriented representation of filters, such as the BlockCompiler, the regular FIR filter and the one in Fig. 6.29 may be just different detailed implementations encapsulated in a single object abstraction. The compiler can modify the code according to the context of use as a regular filter or a reflectance filter.

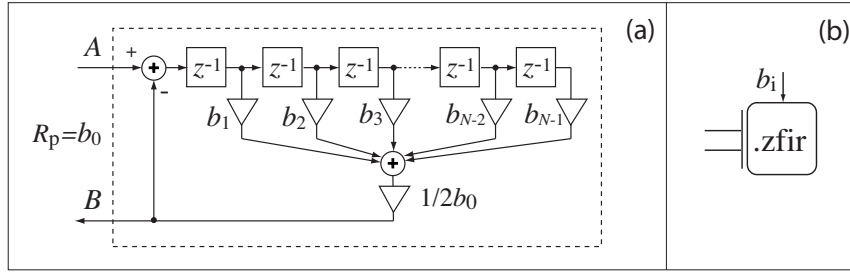


Figure 6.29: (a) One-port block with wave port for a given polynomial impedance of Eq. (6.101) to realize the reflectance of (6.104) and port resistance of (6.103). (b) Symbol used for Block-Compiler.

### 6.8.2 Impedance given in rational form

If the impedance  $Z(z)$  is given as a rational expression

$$Z(z) = \frac{\sum_{i=0}^{N-1} b_i z^{-i}}{1 + \sum_{i=1}^{M-1} a_i z^{-i}} \quad (6.105)$$

then the reflectance  $H(z)$  becomes

$$H(z) = \frac{\frac{\sum_{i=0}^{N-1} b_i z^{-i}}{1 + \sum_{i=1}^{M-1} a_i z^{-i}} - R_p}{\frac{\sum_{i=0}^{N-1} b_i z^{-i}}{1 + \sum_{i=1}^{M-1} a_i z^{-i}} + R_p} \quad (6.106)$$

A T-port without delay-free reflection is possible by selecting

$$R_p = b(0) \quad (6.107)$$

$$H(z) = \frac{\frac{1}{2R_p} \sum_{i=1}^{N-1} b_i z^{-i} - \frac{1}{2} \sum_{i=1}^{M-1} a_i z^{-i}}{1 + \frac{1}{2R_p} \sum_{i=1}^{N-1} b_i z^{-i} - \frac{1}{2} \sum_{i=1}^{M-1} a_i z^{-i}} \quad (6.108)$$

Expression (6.108) leads to an IIR-like realization shown in Fig. 6.30 that is only minimally modified from using the rational impedance expression as an IIR filter.

### 6.8.3 Second order sections

x

### 6.8.4 Lattice formulations

x

### 6.8.5 Impedance given in parallel form

Parallel connection of second-order terms is particularly useful due to its numerical robustness compared to direct-form realizations. It is also a natural choice for modal decomposition of

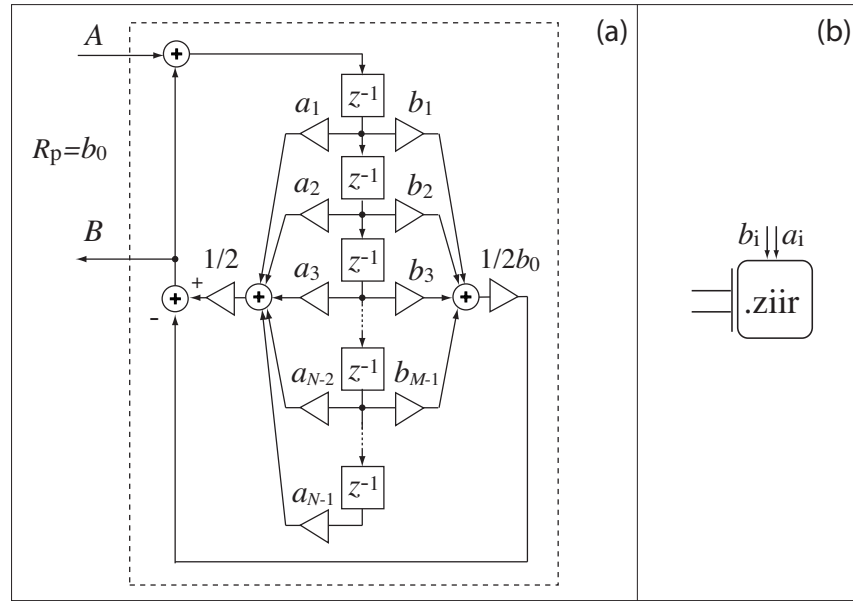


Figure 6.30: (a) One-port block with wave port for a given rational impedance expression of Eq. (6.105) to realize the reflectance of (6.108) and port resistance of (6.107). (b) Symbol used for BlockCompiler.

driving-point impedances/admittances. Impedance can be specified as a sum of a constant term and second-order rational forms by

$$Z(z) = b_{0,0} + \sum_{n=1}^N \frac{b_{0,n} + b_{1,n}z^{-1} + b_{2,n}z^{-2}}{1 + a_{1,n}z^{-1} + a_{2,n}z^{-2}} \quad (6.109)$$

Reflectance  $H(z)$  becomes

$$H(z) = \frac{b_{0,0} + \sum_{n=1}^N \frac{b_{0,n} + b_{1,n}z^{-1} + b_{2,n}z^{-2}}{1 + a_{1,n}z^{-1} + a_{2,n}z^{-2}} - R_p}{b_{0,0} + \sum_{n=1}^N \frac{b_{0,n} + b_{1,n}z^{-1} + b_{2,n}z^{-2}}{1 + a_{1,n}z^{-1} + a_{2,n}z^{-2}} + R_p} \quad (6.110)$$

By the following choices the resulting wave port is of T-type without immediate reflection:

$$R_p = \sum_{i=0}^N b_{0,i} \quad (6.111)$$

$$H(z) = \frac{\frac{1}{2R_p} \sum_{i=1}^N \frac{b'_{1,i}z^{-1} + b'_{2,i}z^{-2}}{1 + a_{1,i}z^{-1} + a_{2,i}z^{-2}}}{1 + \frac{1}{2R_p} \sum_{i=1}^N \frac{b'_{1,i}z^{-1} + b'_{2,i}z^{-2}}{1 + a_{1,i}z^{-1} + a_{2,i}z^{-2}}} \quad (6.112)$$

$$b'_{k,i} = b_{k,i} - b_{0,i}a_{k,i} \quad (6.113)$$

Figure 6.31 shows how this can be implemented with minimal changes to the direct parallel implementation to  $Z(z)$  as a parallel filter. A special case of particularly straightforward realization is when  $b_{0,i} = 0$ , because then  $b'_{1,i} = b_{1,i}$ ,  $b'_{2,i} = b_{2,i}$ , and  $R_p = b_{0,0}$ .

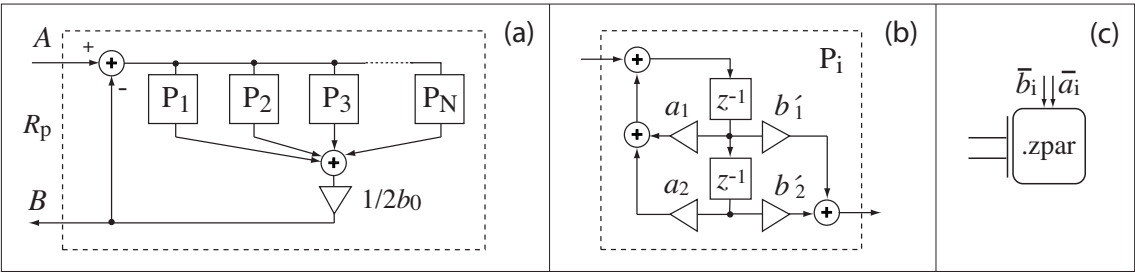


Figure 6.31: (a) One-port block with wave port for impedance given as parallel connection of second-order terms, Eq. (6.109). (b) Realization of single second-order section without delay-free path. (c) Symbol used for BlockCompiler.

6.8.6 Impedance given in cascaded form ???

c

6.9 Conversion of state-space forms to W-elements

x

x



# Chapter 9

## Time-Variant and Nonlinear Modeling

Nonlinear and time-variant systems bring several complications to discrete-time modeling, compared to LTI systems. DSP with uni-directional data flow and without feedback loops may already be difficult in nonlinear cases (Section ??) due to aliasing, but the inherent bi-directionality of interaction and feedbacks in physics-based models adds to this complexity. On the other hand, systematic treatment of energetic behavior in physical systems, such as proper dealing with passivity, can help in managing highly complex nonlinear models [57, 66, 60, 67, ?].

In this chapter the methodology of constructing successful nonlinear and time-varying models is studied from a practical point of view. While there are many approaches to nonlinearity and deep theoretical studies available [?], the goal here is to cover typical cases in physics-based modeling and to demonstrate by numerical simulations how such models work, as well as what are the practical problems with such models.

This chapter begins with definitions of concepts that are important in non-LTI systems, followed by a discussion of general problematic in time-varying and nonlinear modeling. Regarding different element types, resistive nonlinearities are discussed first because of their relative simplicity; this is divided into parameter iteration methods and root element design. Nonlinear reactances are discussed in a similar way, first the parametric iteration method and then the special case of mutator-based reactances with I-port. Spatially distributed nonlinearities are characterized next. Finally the question of global interactions through controlled sources is discussed. These cases are often related to LTI modeling, not only to nonlinearities, but the problems faced are similar to nonlinear modeling.

### 9.1 General aspects of non-LTI modeling

#### 9.1.1 Concepts

In order to understand the basic concepts in the sense they are used here, the following systems terminology is defined:

- **LTI (linear and time invariant) systems** were defined and discussed briefly in Section ??. In such systems the principle of superposition is valid, and no new signal frequencies appear in addition to the frequencies in excitation signals. LTI systems are represented by state variables and constant parameters.

- **Parametric control** brings time variance to the parameters of a model. If the control of a given parameter is independent of the internal state variables of the system, no parametric feedback takes place and therefore no related delay-free loop problems appear.
- **Slow parametric changes due to dependence on state variables** is a limit case between parametric control and true nonlinearity. Slow in this context means frequencies lower or much lower than the signal frequencies of the state variables.
- **(Essential) nonlinearities** can be interpreted as cases where system parameters vary rapidly along with state variables, although in such cases it may be difficult or even impossible to make distinction between parameters and state variables.
- **Smooth vs. hard nonlinearities:** In a smooth nonlinearity the changes in parameters or system properties are gradual, while hard nonlinearities reveal more abrupt discontinuities in parametric behavior. Smooth nonlinearities or parametric controls are typically easier to model successfully than hard nonlinearities.
- **Signal processing vs. physical nonlinearities:** In signal processing (Section ??) nonlinearities are dealt with as transfer properties while in physics-based modeling they are defined as constraints between physical variables. This makes an essential difference conceptually and in the way they are realized.
- **Passive vs. lossless nonlinearities:** When a physical element or subsystem does not produce energy, it is called passive. As a limit case it is lossless if no energy is lost except through port connections to external system. Passivity is an important property to realize stable system models.
- **Resistive vs. reactive nonlinearities:** Resistive elements do not store energy. Therefore their parameter values can be changed without energy preservation considerations. In reactive elements (such as inductors and capacitors) the energy preservation has to be dealt with more rigor. Parametric control of a reactive element may produce, dissipate, or preserve energy, depending on case (see ??).
- **Spatially distributed nonlinearities** are often due to changes in wave propagation, such as varying sound velocity or spatial change in interaction topology. Nonlinearities in digital DWG and FDTD models are of special interest here, but any model for spatially distributed systems may bring up such questions.

### 9.1.2 Port resistance propagation

When the port resistance of an element attached to a WDF adaptor is changed, it will have a global effect within the adaptor and possible further adaptors attached. The propagation of port resistances should happen within the same sample period in the adaptor network. Notice that in a tree structure made by adaptors (Section 6.5.1, Figs. 6.18 and 6.19) the propagation of port resistances flows only towards the root of the network. Therefore it is advantageous to place all time-varying and nonlinear elements as close to the root element as possible and ‘isolate’ the LTI elements by connecting them to separate adaptors further away from the root element so that the need of parameter updating is minimized. Of special interest is using a time-variant or nonlinear element with a I-port as a root element because then it does not have any effect on port resistances elsewhere.

### 9.1.3 Port resistance computation

In parametric control of WDF elements the port resistances can be computed explicitly from the given element parameter values. On the contrary, in nonlinear elements the port resistance is dependent on the state variables. They depend on each other in a way that usually results in implicit equations, i.e., delay-free loops. In such cases there are basically two methods to compute new values: iteration and prediction (extrapolation).

The port resistance  $R_p$  of a nonlinear element is typically expressed as a function of K-variables, for example by  $f(u, i) = 0$  in the electrical domain (in the linear case  $u - Ri = 0$ ). Another level of implicitness comes from the need to use W-variables  $a$  and  $b$  for port variables. Therefore in a general case port resistances are approximated by iterative solution of the implicit equations. The convergence needs to be examined for each case, and the iteration can be stopped after a given number of steps or when the convergence is good enough. Notice that when there are several nonlinearities involved in the network, iteration should cover the effect of all of them.

There are many ways to realize parameter iteration. The simplest one is to use a port resistance value  $R_p$  computed from the state variables of the previous sample instant:  $R_p(n+1) = f[u(n), i(n), a(n), b(n)]$ . It follows, however, that the energetic properties of the network are invalidated compared to the continuous-time system to be modeled. In a less problematic case this means just decreased simulation accuracy, while in a severe case it leads to instability of simulating a physically stable system. More advanced iteration of the port resistances can alleviate such problems, but often it is difficult to guarantee stability and accuracy without practical experimentation.

Port resistance prediction (extrapolation) [?] is an approximate method to estimate the new value(s) of port resistance(s). By knowing a set of previous values and assuming smoothness of parametric changes the new value(s) can be predicted using a linear or nonlinear predictor. The simplest predictor is just using the value of port resistance based on the previous values of state variables, i.e., actually no prediction at all, which is also equal to no iteration at all. While approximative parameter prediction itself can be stable, it does not necessarily guarantee stability (passivity) of the network.

### 9.1.4 Aliasing and frequency warping

Nonlinearities create new frequency components that are not included in the excitation of the network. In audio applications of interest here a dangerous consequence is that such components falling beyond the Nyquist frequency will mirror back to the baseband, and they can be perceptually highly disturbing. Such *aliasing distortion* should be kept at a level low enough. In some cases it is possible to shape the nonlinearities so that aliasing is eliminated [?], but in most cases oversampling (increased sample rate) and limiting of excitation bandwidth is the only practical means to counteract aliasing.

As with linear WDF elements, frequency warping due to the bilinear mapping, see Eq. (6.32) and Fig. 6.8, is present also in nonlinear elements. This can be counteracted by oversampling or by prewarping techniques.



### 9.1.5 Iteration per sample vs. oversampling

In cases requiring parameter iteration there are two choices available: (a) iteration to convergence with desired accuracy per each sample instant or (b) oversampling by a factor high enough to decrease artifacts. These have different advantages and drawbacks:

- *Iteration per sample* is the ‘mathematically correct’ way of realizing the model simulation with a given sample rate. The advantage is that convergence can be inspected and iteration be stopped when the accuracy is good enough. From an audio point of view two problems may remain unsolved, however. The aliasing effect due to nonlinearities is not removed, and the frequency warping problem needs to be solved separately, if it is essential in a given case.
- *Oversampling* can be seen as an ‘implicit iteration’ when looking at the base band. From a ‘mathematically correct’ viewpoint oversampling is only motivated when it approaches infinity, i.e., a continuous-time simulation. From an audio viewpoint, however, it is attractive as a straightforward method of just specifying a higher sample rate. It also brings the remarkable advantages of reducing aliasing and frequency warping of the bilinear mapping rapidly with increased sample rate.

## 9.2 Parametric control of WDF elements

WDF elements are the most essential components for lumped element models. In many cases the parameter values of such components should be controllable, and in nonlinear cases the signals (state variables) in the system need to change the parameter values. In this section we discuss different ways to make WDF element parameters controllable. This information will be utilized in the subsequent chapters for nonlinear modeling. The three main methods discussed here are (a) direct control of port resistance  $R_p$ , (b) use of controllable root elements that have a I-port, and (c) control using an ideal transformer. The discussion will cover resistances, resistive sources, and reactive elements separately.

### 9.2.1 Time-varying resistances

Parametric control of resistances is relatively easy because they do not store energy. Therefore in changing a resistance value, as far as the port resistance remains non-negative, there is no need to consider further what happens with the energetic state of the element.

#### Direct control of WDF resistance

The WDF resistance was discussed in Section 6.2.1, where it was given that a reflection-free (T-) port resistance is achieved by selecting the port resistance  $R_p$  to equal the desired resistance value  $R$ . Because the reflected wave  $b = 0$  for any value of incident wave  $a$ , the only consequence of changing resistance value  $R$  is the corresponding change in  $R_p$ . This of course has consequences further in the interconnected network through the adaptors, but the resistance itself is easily realized.

#### Control of WDF resistance as a root element

The special case of resistance with a I-port that can be used as a root element of a WDF network was presented in Section 6.3.1. The advantage of this type is that the other parts of the network

tree are independent of changes is the resistance. Thus the resistance  $R$  can be changed according to Eq. (6.40) and Fig. 6.10 without need to update any other parameter elsewhere. The use of such an element is limited only to a single instance, positioned as the root element.

### Control of resistance using ideal transformer

The impedance level of any port can be changed using an ideal transformer with a variable turns ratio  $1:N$  (Section 6.6.1). As described by Eq. (6.68) and Fig. 6.21, the transformed port resistance  $R_x (= R_{p2})$  at the secondary port of the transformer, the primary port being connected to resistor  $R (= R_{p1})$ , is

$$R_x = N^2 R \quad (9.1)$$

which implies that  $R_x$  is obtained by turns ratio

$$N = \sqrt{R_x/R} \quad (9.2)$$

No particular advantage is achieved compared to direct control of  $R_p$  due to additional complexity of computation.

## 9.2.2 Time-varying resistive sources

Time-varying resistive sources can be analyzed in a way similar to the resistance. For example in a voltage source with internal resistance  $R_i$ , as presented in Section 6.2.3 (Fig. 6.4), the driving voltage  $E$  is an independent signal variable that can be varied freely, and internal resistance  $R_i$  can be changed as any resistance.

A resistive source can be transformed also by an ideal transformer to a new source with driving voltage  $N$  times  $E$  and internal resistance  $N^2$  times  $R_i$ , where  $N$  is the turns ratio of the transformer. No specific advantage is achieved, however, compared to direct control of the port resistance.

## 9.2.3 Time-varying reactances

Reactive elements, such as capacitances and inductances in the electrical domain, need more consideration than the resistive elements. This is due to the fact that reactances are energy storages, where the change of parameter value may or may not change the energy content of the element, depending on the case at hand. For example with passive and lossless nonlinearities discussed in the sections below the change in a parameter value should not increase the energy content of the element. In this subsection we will study the behavior of capacitances controlled in different ways.

### Energy behavior of a capacitor with port resistance control

The simplest idea to make a time-varying capacitor is to control the port resistance according to Eq. (6.24), i.e.,  $R_{p,n} = T/(2C_n)$ , where  $T$  is the sample interval,  $C$  is the capacitance, and  $n$  is the sample index. The energetic behavior of such a controllable capacitance can now be analyzed in the following way.

Figure 9.1 shows a WDF capacitor charged to carry voltage  $u$  and with open circuit termination. Because there is no current flowing through the capacitor, Eq. (6.3) implies that

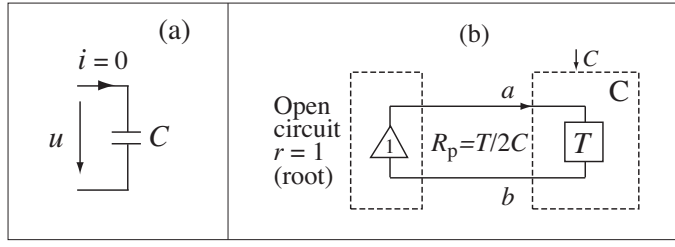


Figure 9.1: (a) Open-circuited capacitor controlled by port resistance  $R_p$  and charged initially with voltage  $u$  (b) WDF network for simulation.

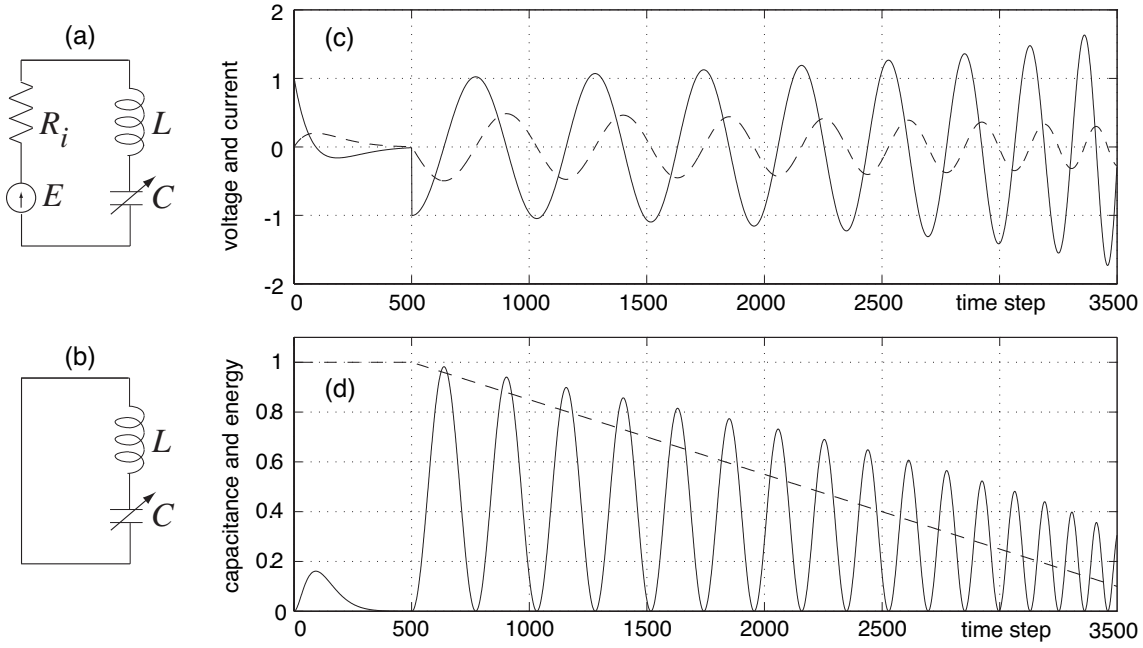


Figure 9.2: Effect of a time-varying capacitance in an LC circuit (b) when the capacitance is controlled directly by changing port resistance. The circuit is first charged (a) and then the voltage source is removed (short-circuited) and the capacitance is varied linearly from 1 mF to 0.1 mF. (c) Inductor voltage (solid line) and current (dashed line). (d) Relative inductor energy (solid line) and capacitance (dotted line).

instantaneous wave variable values  $b = a$  are equal and therefore the voltage  $u = 2a$ . The energy  $E$  of the capacitor is

$$E_n = \frac{1}{2}Cu^2 = 2Ca^2 \quad (9.3)$$

If the capacitance is now changed from time index  $n$  to the next one  $n + 1$  by  $C_{n+1} = kC_n$  through  $R_{p,n+1} = R_{p,n}/k$ , then the energy change, interpreted as above, is

$$E_{n+1} = 2(kC)a^2 \rightarrow \frac{E_{n+1}}{E_n} = k \quad (9.4)$$

On the other hand, the power flow at the port according to Eq. (6.4) is  $W = a^2/R_p - b^2/R_p = 0$  for any instantaneous value of  $R_p$ , since  $b = a$ . This shows that the energetic interpretation of controlling the capacitance in this way is not consistent.

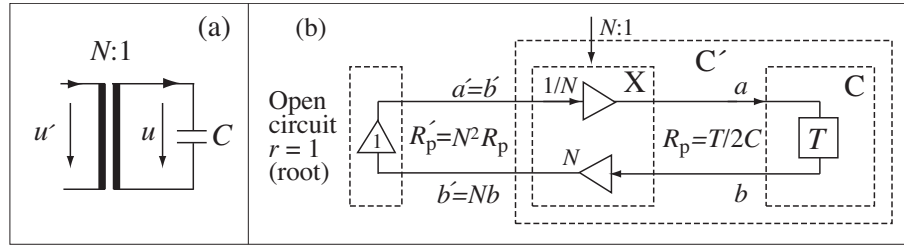


Figure 9.3: (a) Open-circuited capacitor controlled by variable ideal transformer and charged initially with voltage  $u$  and (b) WDF network. Here  $C$  = constant capacitance,  $X$  = ideal transformer with turns ratio of  $N:1$ ,  $C'$  = transformed capacitance,  $u'$  = transformed port voltage.

Another way to study the energetic behavior of such a capacitance is to make a series LC resonator ( $L = 4$  mH,  $C = 1$  mF initially) with controllable capacitance and to observe the energy of the inductor with time-varying capacitance. Figure 9.2 shows the result of such simulation. The circuit is first precharged from a series voltage source of 1 Volt and  $2 \Omega$  internal resistance (sample range 0...500). Then the source is removed (short-circuited) and the capacitance is varied linearly from 1.0 to 0.1 mF. The consequence is that the resonance frequency will increase by factor  $\sqrt{10}$ , the peak value of inductance voltage grows, and the peak value of current decreases. As the lower subplot shows, the peak energy of the inductor decreases approximately by the square root of the capacitance.

The analysis shows the inconsistency of energetic behavior of capacitance where the port resistance is controlled directly. If the relative change of capacitance (deviation of  $k$  from 1) is small for each sample step, the method can be useful in practice as will be demonstrated later.

### Time-varying capacitance by transformer control

The ideal transformer (Section 6.6.1) is a non-energetic two-port (i.e., no energy storage) that can map an impedance level to another. Therefore it is a potential device for generalized control of impedances in an energetically sound way. Figure 9.3 depicts the case where a constant-valued capacitance  $C$  is transformed through an ideal transformer with variable turns ratio of  $N:1$  and open-circuited at the left-hand side.

Let us assume that at the moment  $n$  the turns ratio is  $N_n = 1$  so that the capacitance  $C$  is transformer-mapped to capacitance  $C'_n = C$  and the port variables at both ports of the transformer are equal. Now let the turns ratio be changed to value  $N_{n+1}$  for the time index  $n + 1$ . The parameters and state variables of the transformed capacitor  $C'$  become

$$R'_{p,n+1} = N_{n+1}^2 R_p \quad (9.5)$$

$$C'_{n+1} = C/N_{n+1}^2 \quad (\text{see Eq. (6.24)}) \quad (9.6)$$

$$b'_{n+1} = N_{n+1} b_{n+1} = N_{n+1} a_n \quad (9.7)$$

$$a'_{n+1} = b'_{n+1} \quad (9.8)$$

$$a_{n+1} = a'_{n+1}/N_{n+1} = b_{n+1} = a_n \quad (9.9)$$

From Eq. (9.6) it follows that transformer turns ratio

$$N = \sqrt{C/C'} \quad (9.10)$$

yields capacitance value of  $C'$  from a fixed capacitance  $C$ .

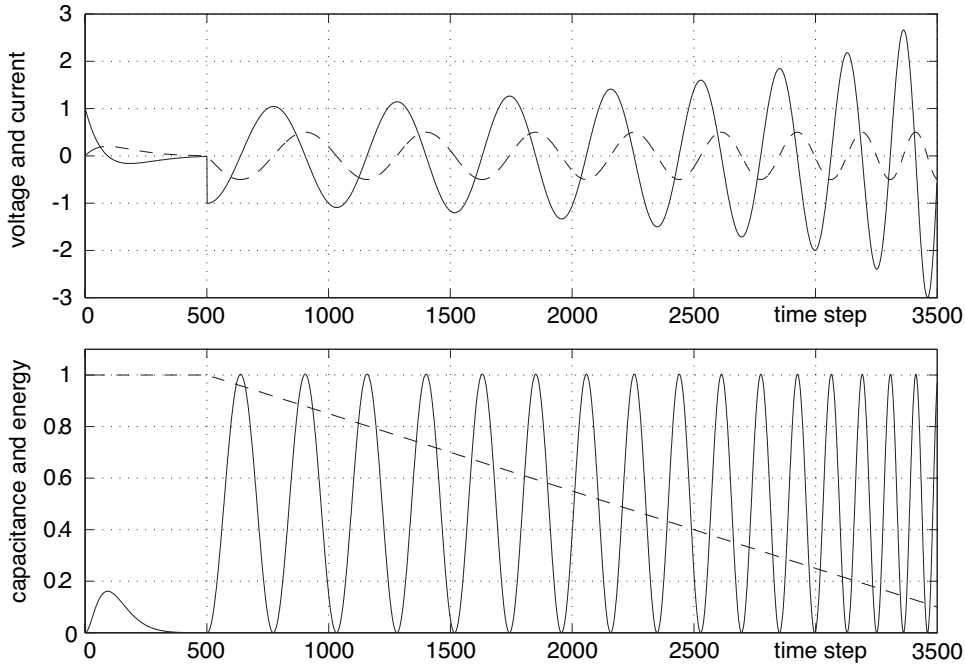


Figure 9.4: Effect of a time-varying capacitance in an LC circuit when the capacitance is controlled by an ideal transformer. Top: inductor voltage (solid line) and current (dashed line). Bottom: relative inductor energy (solid line) and capacitance (dotted line). The circuit is first charged and then the capacitance is varied linearly from from 1 mF to 0.1 mF.

Eq. (9.9) shows that the state of the capacitor  $C$  is not changed when  $N$  and thus  $C'$  is varied. The original energies of the capacitance  $C$  as well as of  $C'$ , as analyzed above in Eq. (9.3), are  $E_n = E'_n = 2Ca_n^2$ . The energy of the transformed capacitance at time index  $n + 1$  becomes

$$E'_{n+1} = \frac{1}{2} \frac{C}{N_{n+1}^2} (2N_{n+1}a_n)^2 = 2Ca_n^2 \quad (9.11)$$

which shows that the energy remains unaltered for any change in turns ratio and therefore for a change in transformed capacitance value. This is a very valuable feature when realizing nonlinear reactances, where the capacitance (or inductance) is dependent on state variables so that lossless elements are obtained.

Figure 9.4 presents the simulation results for an LC circuit with transformer-controlled capacitance, otherwise similar to the direct port resistance control above. Now the change in capacitance has the same effect on resonance frequency, but with decreasing capacitance the peak voltage grows more rapidly and the peak current remains constant. As expected, the peak energy also remains constant, which is in line with the analysis of the open-circuited capacitor above.

### Time-varying inductance through transformer control

By following a similar analysis it is possible to study the behavior of a time-varying inductance realized by controllable ideal transformer. A dual model is needed where the inductance carrying a current is short-circuited. In that case the inductance value is controlled through formula

$$N = \sqrt{L'/L} \quad (9.12)$$

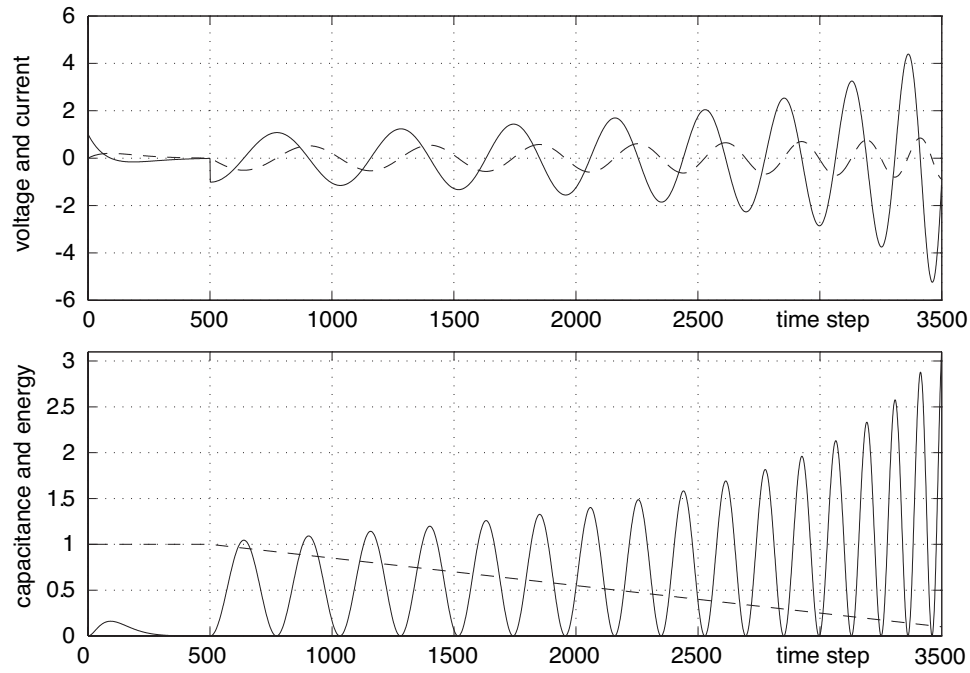


Figure 9.5: Effect of a time-varying capacitance in an LC circuit when the capacitance is realized by a mutator. Top: inductor voltage (solid line) and current (dashed line). Bottom: relative inductor energy (solid line) and capacitance (dotted line). The circuit is first charged and then the capacitance is varied linearly from 1 mF to 0.1 mF.

### Time-variant reactances by mutators

Section 6.6.6 described how a mutator can be used to map a resistance into a capacitance or an inductance. Compared to the transformer-based capacitor above, the energetic behavior is more involved to analyze, because the mutated capacitor cannot be terminated with an open circuit (or else  $R_p \rightarrow \infty$  and  $(1 - r) \rightarrow 0$  in Eq. (6.83)).

The behavior of a mutator-based capacitance can be simulated numerically for the same LC-circuit case as direct control and the transformer-based control above. Figure 9.5 plots the resulting set of curves. The resonance frequency behaves in similar way as the two previous cases (Figs. 9.2 and 9.4), but both the peak voltage and current grow more rapidly with decreasing capacitance. The peak energy grows also with decreasing capacitance value. Therefore this solution is not as good for implementing nonlinear passive reactances as the transformer-based method.

#### 9.2.4 Power normalization in time-variant networks

In Section 6.1.2 it was noticed that WDF port power flow  $W$  can be made independent of port resistance change,  $W = ui = \tilde{a}^2 - \tilde{b}^2$ , by using (instantaneous) power normalized waves

$$\begin{cases} \tilde{a} = (u + R_p i) / (2\sqrt{R_p}) \\ \tilde{b} = (u - R_p i) / (2\sqrt{R_p}) \end{cases} \Leftrightarrow \begin{cases} u = (\tilde{a} + \tilde{b})\sqrt{R_p} \\ i = (\tilde{a} - \tilde{b})/\sqrt{R_p} \end{cases} \quad (9.13)$$

where variables with tilde refer to a power-normalized waves. These wave variables are also called “power waves” [?, 68, ?], although in fact they are rather “square root power waves”.

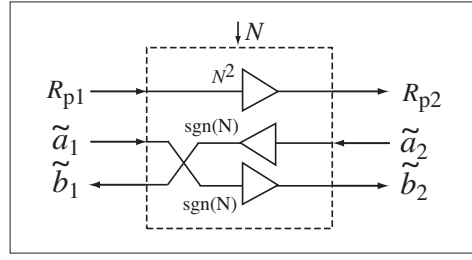


Figure 9.6: Power-normalized ideal transformer.

### Power-normalization at wave ports

The relation between power-normalized and physically normalized variables at wave ports is

$$a = \sqrt{R_p} \tilde{a} \quad \text{and} \quad b = \sqrt{R_p} \tilde{b} \quad (9.14)$$

Notice that in (voltage and current) sources there is need of scaling by  $1/\sqrt{R_p}$ , for example for the voltage source to apply

$$B = E/(2\sqrt{R_p}) \quad (9.15)$$

instead of Eq. (6.16).

These scalings together show that for LTI models with constant  $R_p$  the power normalization and the physical normalization yield equivalent behavior in terms of K-variables. For such LTI WDF-models there is no reason to use power normalization, because it only brings the numerically somewhat slow computation of  $\sqrt{R_p}$ . On the contrary, in time-varying and nonlinear models it may be advantageous or necessary to use power normalization in order to guarantee passivity of a network.

### Power-normalization of ideal transformers

While the one-port elements have been scaled in a simple way in power-normalization, elements with at least two non-equal port resistances need more consideration. For the ideal transformer (Section 6.6.1) the power flow in each direction must have value of unity, so that

$$\tilde{b}_1 = \text{sgn}(N) \tilde{a}_1 \quad (9.16)$$

$$\tilde{b}_2 = \text{sgn}(N) \tilde{a}_2 \quad \text{and} \quad (9.17)$$

$$N^2 = R_{p,0}/R_{p,1} \quad (9.18)$$

where  $\text{sgn}(N)$  is the sign of the turns ratio  $N$ , having value of +1 or -1, see Fig. 9.6.

### Power-normalized adaptors

For power normalization of a network, adaptors need to be normalized as well. This is done by replacing voltage wave scattering coefficients of the parallel adaptor in Eq. (6.52) by power scattering forms

$$\gamma_n = \tilde{\gamma}_n^2 \quad \rightarrow \quad \tilde{\gamma}_n = \sqrt{2G_n / \sum_{i=1}^N G_i} \quad (9.19)$$

where  $G_i = 1/R_i$  are adaptor port admittances. It can be seen easily now that  $\sum_{n=0}^{N-1} \gamma_n^2 = 2$ . A reflection free T-port can be made in a similar way as in Section 6.4.

For a power-normalized series adaptor, instead of Eq. (6.59), we can write

$$\tilde{\gamma}_n = \sqrt{2R_n / \sum_{i=1}^N R_i} \quad (9.20)$$

where  $1/R_i$  are adaptor port resistances, and reflection-free ports are made as in Section 6.4.

Notice that in a network containing both non-LTI and LTI elements it is possible to collect non-LTI elements closest to the root element and make only that part power-normalized, while the LTI-part is not power-normalized. This minimizes the computational load of the model.

## 9.3 Nonlinear resistances

Nonlinear resistance means that the dependence between voltage  $u$  and current  $i$  does not follow Ohm's law  $u = Ri$ , but must be expressed in one of the following forms:

$$u = f_R(i) \quad \text{explicit} \quad (9.21)$$

$$i = f_G(u) \quad \text{explicit} \quad (9.22)$$

$$f_r(u, i) = 0 \quad \text{implicit} \quad (9.23)$$

For a resistive nonlinearity the following should also hold:

$$u = 0 \quad \leftrightarrow \quad i = 0 \quad (9.24)$$

In wave-based modeling the variables used are waves  $a$  and  $b$  instead of K-variables ( $u$  and  $i$  in the electrical domain). In a linear resistance with an T-port (Section 6.2.1)  $b = 0$  always, so there is no signal computation within such a resistor block. In a similar case for a nonlinear resistor the port resistance  $R_p = R$  needs to be controlled so that the characteristics of the form in Eqs. (9.21)–(9.23) are realized. The basic problem is that the new values state variables  $\{u, i, a\}$  are not known yet to compute  $R_p$ , which is however needed to compute the state variables. This means that iteration is needed to solve the implicit equation, which is discussed in the next subsection.

For a resistance with a I-port (Section 6.3.1) the computation of function  $b = f(a)$  is needed.

### 9.3.1 Parametric iteration of resistance

Based on the above discussion we can conclude that some sort of port resistance iteration or prediction is needed in nonlinear models. The delayed parameter updating, possibly combined with oversampled computation, is the simplest technique that in many cases works relatively well. In some cases, for example with high accuracy requirements, more advanced methods may be required. In the next section we will explore simple iterative techniques for different WDF elements using numerical examples.

#### *Example: Vacuum tube rectifier*

As the first example of nonlinear resistors the behavior of a vacuum tube rectifier is simulated. The voltage-current characteristics for positive voltages can be approximated [?] by

$$i = k u^{3/2} \quad \leftrightarrow \quad u = (i/k)^{2/3} \quad (9.25)$$



For negative voltages the current is approximately zero. From Eq. (9.25) the resistance characteristics  $R = u/i$  can be derived as a function of  $u$  or  $i$  as

$$R(i) = k^{-2/3} i^{-1/3} \quad \text{or} \quad R(u) = k^{-1} u^{-1/2} \quad (9.26)$$

Figure 9.7(a) plots the U/I-characteristics of the tube along with a load line for a load resistor and source voltage according to Fig. 9.8. Figure 9.7(b) shows the tube resistance  $R(u) = u/i$  as a function of voltage over the tube. For voltages  $u \leq 0$  the resistance becomes infinite. For practical simulations a large finite value needs to be used instead for numerical reasons (1 M $\Omega$  in this case).

The curves in 9.9 reveal some oscillation of the state variables as well as the nonlinear resistance value after an abrupt change in the driving voltage. For a positive voltage step the resistance oscillates some 4-8 time steps depending on the required convergence, and for the negative voltage step there is a single time step of delay to correct the resistance value. The two ways to reduce the oscillation are (1) to run full iteration for required accuracy for each time sample and to use the converged value or (2) to limit the bandwidth (rate of change) of the driving signal and apply oversampling if needed.

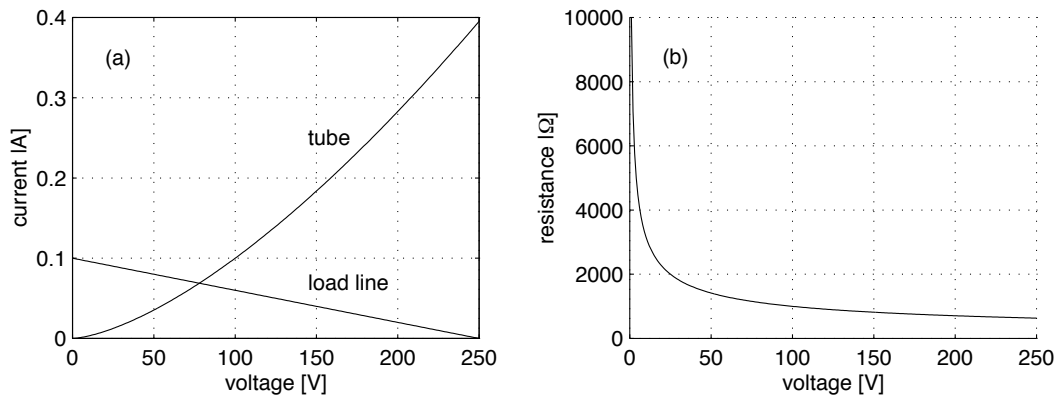


Figure 9.7: (a) Vacuum tube characteristics for  $k = 0.0001$  and load line for  $R_L = 2500 \Omega$ ,  $E = 250$  V, (b) tube resistance as a function of voltage.

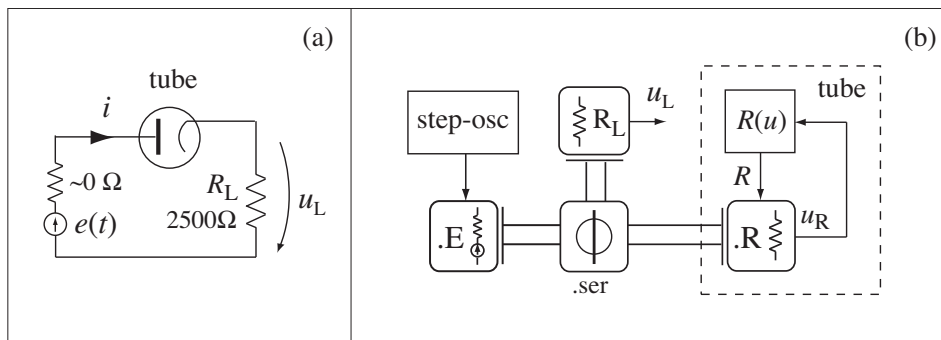


Figure 9.8: (a) Circuit with nonlinear resistance (tube diode rectifier), (b) WDF network for simulation.

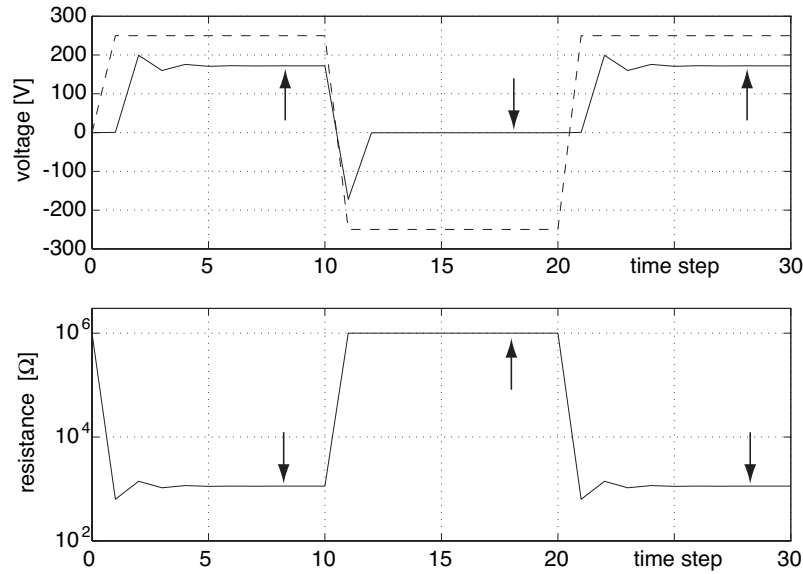


Figure 9.9: WDF Simulation of the circuit in Fig. 9.8 for nonlinear resistor (tube rectifier) as functions of time step: (top) voltage  $u_L$  across load resistance (solid line) and source voltage step function (dashed line), (bottom) variation of the nonlinear resistance. Arrows point the values obtained in 8-step iteration per sample with 8 times lower sample rate.

### 9.3.2 Nonlinear resistors with I-port

x

v, i -z a, b

piecewise linear curves

table lookup and interpolation

function approximation

x

x

## 9.4 Nonlinear capacitance

The realization of a linear lumped capacitance as a WDF element was discussed in Subsection 6.2.5. In this subsection we will study different possibilities to approximate nonlinear capacitances in the framework of wave digital filter elements.

### 9.4.1 Parametric iteration of a capacitor

A straightforward approximation of a nonlinear capacitance (and compliance in the mechanical domain) is obtained by making the capacitance parameter (and thus the port resistance) dependent on the voltage (force) over the element. Let us assume that the nonlinear characteristic curve of a capacitor is given by  $q(u)$ , where  $q$  is the charge and  $u$  is the voltage over the element. Discrete-time charging of the capacitor can be approximated by

$$q(n_t) = q(0) + \sum_{n=1}^{n_t} \frac{dq(u)}{du} \Delta u(n) \quad (9.27)$$

The derivative  $dq(u)/du$  is differential capacitance and can be marked as

$$C_d(u) = \frac{dq(u)}{du}, \quad \text{for the linear case: } C_d = C = q/u = \text{constant} \quad (9.28)$$

Now the WDF capacitor in Subsection 6.2.5 can be used by a modification of changing the capacitance parameter value according to changes in  $C_d$ , which means changing the port resistance by

$$R_p = \frac{T}{2C_d} \quad (9.29)$$

As with the nonlinear resistor above, the new port resistance is not computable based on new state variables, so that iterative solution or previous port resistance value and oversampling is needed for improved accuracy.

### Example: Nonlinear compliance of a spring

The compliance of springs is often dependent on the displacement from equilibrium state. A good example is the suspension of a loudspeaker diaphragm, where the surrounding suspension and the spider make a compliance that starts to saturate for large displacements. We can describe such characteristics for example by analytic formula

$$x(F) = \tanh(F), \quad \rightarrow \quad C_{d,m} = \frac{d}{dF} x(F) = \frac{1}{\cosh^2(F)} \quad (9.30)$$

where  $F$  is force applied,  $x$  is displacement, and  $C_{d,m}$  is the differential mechanical compliance of the spring. Figure 9.10 plots the behavior of such compliance when a linearly growing force is applied to it. Subplot (a) shows the force and displacement as functions of time. Linearly growing force results in a displacement according to the tanh function. The lower subplot shows the differential compliance  $C_{d,m}$  (solid curve) and the ‘integral compliance’  $C_m = x/F$  (dotted curve) as functions of the applied force.

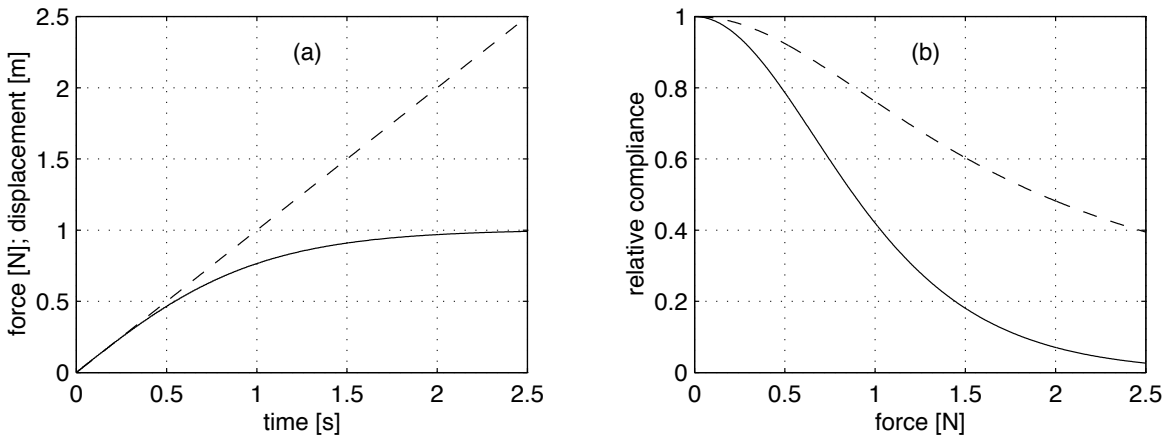


Figure 9.10: Nonlinear compliance of a spring: (a) displacement of the spring from equilibrium (solid curve) and the force applied (dashed curve) functions of time; (b) differential compliance (solid curve) and ‘integral compliance’  $C_m = x/F$  (dotted curve) as functions of applied force.

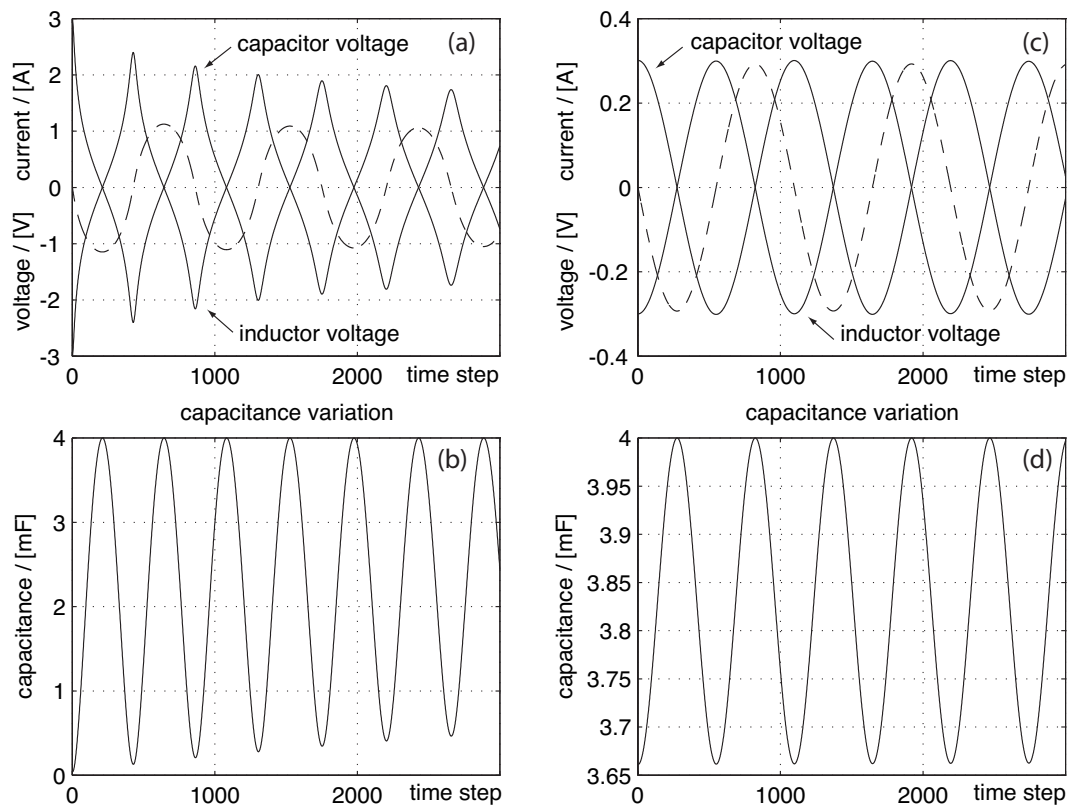


Figure 9.11: Behavior of a series LC circuit with nonlinear capacitor with initially charged capacitor. (a) Large-signal behavior of voltages over capacitor and inductor (solid lines) as well as circuit current (dashed line), (b) variation of capacitance value. (c) and (d) Corresponding small-signal behaviors.

Dynamic behavior of nonlinear capacitor

x

- x
- x
- x
- x
- x
- x
- x
- x
- x

9.5 Nonlinear inductance

x

9.5.1 Nonlinear capacitors and inductors with I-port

x

## **9.6 Distributed nonlinearities**

x

## **9.7 Non-local interaction and controlled sources**

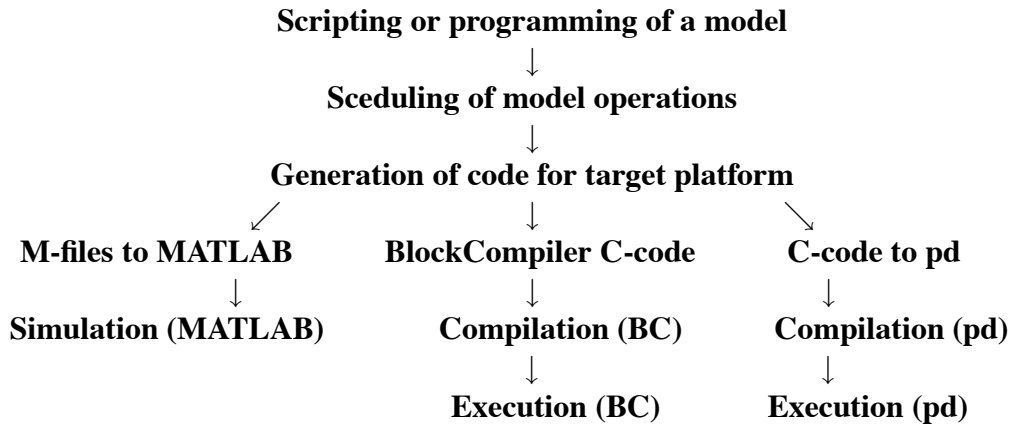
x

# Chapter 11

## Introduction to BlockCompiler (BC)

*BlockCompiler*, shortly BC, is a software tool developed for multi-paradigm physics-based modeling for discrete-time simulation and real-time synthesis. In addition, it supports non-physical modeling using DSP algorithms. It is block-based (object-based) so that computational models are built by connecting physical blocks through their ports or DSP blocks through their input and output terminals.

BC can be characterized as a model-building tool, code generator, compiler, and execution engine. Depending on which platform is used for final execution of the models, BC either runs them within BC itself or they are first exported to another environment, such as Matlab [81] or pd (PureData) [82], for execution. BC functionality, starting from model building to final simulation and execution, is the following:



The advantage of the code generator approach is that BC can be quite easily adapted to other execution environments by writing a new code generation part, while the model-building part remains the same.

BC is designed for time-domain sample-by-sample computation, although it supports also DSP computation in data blocks (buffers). The sample-by-sample execution is a necessity in physics-based models with two-directional interaction, since the preceding sample step values are needed for the computation of the subsequent values, and thus data-buffer sizes longer than one cannot be used.

In some aspects BlockCompiler resembles many other software tools that are designed for modeling, simulation, or sound synthesis. Among them are for example pd (PureData) [82], MaxMSP [83], Simulink [81], LabView [84], Mustajuuri [85], and PatchWork [86]. They are visual programming languages, where algorithms can be specified on the computer screen by

interconnecting computational blocks through their input and output terminals. This is an attractive feature for non-programmers (in the traditional sense). Text-based programming and scripting is, however, very powerful and avoids many of the problems that appear when trying to visualize complex algorithms and data structures. Therefore BlockCompiler is in the first hand based on textual programming and scripting, with no visual programming interface presently, although such one could naturally be added. The computational diagrams in this document used in describing BC models are for illustrative and learning purposes only, they are not (at least yet) supported by the BC user interface.

BlockCompiler is implemented in the Common Lisp programming language [87]. Lisp syntax, based on the list notation with quite many parentheses, may look somewhat strange in comparison to more often used languages such as C and Java. It takes some effort to become a power user and programmer in Lisp, but simple scripting that is needed to build fairly advanced models in BC is easy to learn<sup>1</sup>.

The rest of this chapter first introduces a bit of Lisp syntax and presents some basic properties of the BlockCompiler software. Then it describes the usage of BC by a set of relatively straightforward examples. The aim is to give a quick start to the reader in exploring basic physics-based as well as DSP-based models and algorithms. To install and configure BC, consult web page <http://www.acoustics.hut.fi/software/BlockCompiler/>.

## 11.1 Basic Lisp syntax

Lisp syntax is based on lists, i.e., forms made of data entities within enclosing parentheses. The following Lisp expression is an ordered collection consisting of an integer, a floating-point number, a string, a symbol, and (another) list of two integers:

```
(12 12.34 "this is a string" x12 (1 2))
```

Here is another list that works also as an executable form, where the first element is a function that adds the next elements and returns the sum when executed in the Lisp listener:

```
(+ 1 2.34 (+ 1 2)) => 6.34
```

Notice that the addition function (+) can take any number of arguments<sup>2</sup>. Evaluation of a form can be cancelled by preceding it with quote ('), for example when executing

```
'(+ 1 2.34) => (+ 1 2.34)
```

If the first element in a list to be evaluated is not a function, or if the elements following it are not proper arguments to it, an error message results:

```
(a b c)           => error (a not a function)
(+ 1 "string" 2)  => error ("string" not a number)
```

Comment texts in Lisp can be separated from code by one or more semicolons (;), e.g.,

```
;;; This entire line is a comment
(+ 1 2 3) ; This is a comment following an expression
```

<sup>1</sup>Choosing a more common syntax for model building was considered, but it has the disadvantage of reducing the high representational power and flexibility that Lisp has.

<sup>2</sup>The same is true with multiplication (\*), subtraction (-), and division (/). For subtraction, (- x1 x2 x3 ...) means subtracting arguments x2 and x3, etc., from x1. The form (- x1) negates the argument x1. The same principle holds for division.

In Lisp there are many special forms and macros that are useful. Among the most important ones are the `let` and `let*` forms<sup>3</sup>. They are used to name local variables (by a list of lists) that can be referred to within the body of the form. For example in the following form

```
(let* ((a (+ 1 2))   ;; a is local and bound to 3
      (b (+ a 3)))  ;; b is local and bound to 6
      (+ a b))      ;; body form of let*, returns 9
```

variable `a` is bound to 3, variable `b` is bound to 6, and the entire form, when evaluated, returns value 9.

## 11.2 Making blocks and patches in BC

Computational *blocks* in BC are created by *make-functions* that are separated from Lisp functions by a dot in the beginning of the function name, such as `(.add)`. Another difference is that when for example `(.add)` is evaluated, it only makes and returns an adder block but does not compute any addition. Blocks need to be connected to a *patch*, an interconnected network, which will be computed or evaluated only by a separate command as will be discussed below.

A patch can be defined using the `defpatch` form<sup>4</sup>, as is done in the following example:

```
(defpatch px ((x1 (.var 1.2))   ;; BC variable in x1 (1)
              (x2 (.const 2.3)) ;; BC constant in x2 (2)
              (adder (.add)))   ;; BC adder in adder (3)
  (-> x1 adder)                 ;; x1 to 1st input of adder (4)
  (-> x2 (in adder 1)))         ;; x2 to 2nd input of adder (5)
```

Here the first line (1) starts `defpatch` definition, i.e., making of a patch named `px` by declaring a local variable (like in a `let*` form) named `x1`. The value of this Lisp variable will a BC variable<sup>5</sup> of value 1.2. The BC variable is created by function `.var`. The second line (2) makes a BC constant of value 2.3 bound to `x2`. Line (3) makes a BC adder (with two inputs by default), now bound to local Lisp variable `adder`.

The body of the `defpatch` form in the above example specifies interconnections between the terminals of the BC blocks. Line (4) connects the output of `x1` to the first input<sup>6</sup> of the adder using the chaining function `->`. Line (5) connects variable `x2` to the second input of the adder<sup>7</sup>.

The patch described above is a perfectly valid DSP patch in BC, although it is not very useful. To make meaning, more complex patches with proper input and output blocks are needed. For physics-based models, blocks for physical elements are used with port interconnection functions, as will be described later.

The next subsections introduce to using BC in four different ways. In each of them, a patch is created and then simulated in BC or exported to another environment. First, model export to MATLAB/Octave is presented in Subsection 11.2.1. In 11.2.2 the simulation is done in BC

<sup>3</sup>Special form `let` deviates from `let*` in that a local variable cannot be referred to previous ones in the definition list, while in a `let*` for such reference is valid, as the next example shows.

<sup>4</sup>More detailed description of patch creation with different formulations is given in Appendix A.1.1.

<sup>5</sup>First it may be somewhat confusing to keep Lisp entities and BC entities conceptually separate, but the logic should become clear by a bit of exercise.

<sup>6</sup>More specifically, inputs and outputs are referred to by forms like `(in adder 0)` and `(out adder 0)`, but the chaining function `->` automatically finds the first input or output if only the block is referred to.

<sup>7</sup>Notice that indexing in BC starts from 0.



and only the result is exported for visualization. The third case is real-time executions in BC itself in 11.2.3, and finally exporting a patch to Pd for real-time execution is presented in 11.2.4. You may try any of them as far as your BC installation and availability of external applications support them, or you may try only what you are interested in. It is recommended, however, to read through all cases.

### 11.2.1 Exporting models to MATLAB/Octave

In the BC3 version of BlockCompiler, models (patches) can be exported to MATLAB<sup>8</sup> or Octave<sup>9</sup> m-files by evaluating the Lisp form:

```
(to-matlab px)
```

where `px` is the patch to be exported. The directory where the model files will be written must first be declared as instructed in the BC3 installation guide. Also, this director needs to be in MATLAB's list of pathnames to enable calling the exported model scripts. Model export to MATLAB creates two files:

1. `bc_init`, when called in MATLAB, will initialize all variables and functions needed to execute the patch.
2. `bc_step`, when called in MATLAB after `bc_init` has been called, will simulate the model one sample step.

Now the user can use MATLAB (or Octave) to run patch simulation, step by step, with all the support MATLAB provides<sup>10</sup>.

Often there is need to run a simulation by looping a certain number of time steps and then to plot the simulation results. A typical case is to compute and plot an impulse response or related frequency response. In such cases the function `matlab-response` does the exporting as described above, additionally runs simulation, and finally presents the result. As an example, a simple sinewave oscillator is made first by evaluating the script (in file "mdemos/dsp/sinosc")<sup>11</sup>:

```
(defpatch sinosc ((sin (.sin-osc :freq 1000.0 :ampl 1.0))
                  (out (.probe "out"))))
  (-> sin out)) ;;; sine wave 1 kHz amplitude 1.0 to "out"
```

This patch is for generating a sinewave with frequency of 1 kHz and amplitude of 1.0.

The syntax of the the first line needs some explanation. BC blocks can have two types of inputs: *signal inputs* and *parameter inputs*. The basic different is that signal inputs are processed synchronously for each sample step, but parameter inputs may be asynchronous and computed by a different rate or only when needed.

<sup>8</sup>MATLAB® software available from MathWorks Inc., <http://www.mathworks.com/>

<sup>9</sup>GNU Octave is freely redistributable software, mostly compatible with MATLAB, see <http://www.gnu.org/software/octave/>

<sup>10</sup>BC, used this way together with MATLAB, is like a preprocessor that schedules sample-by-sample simulation of the given patch. In complex cases this can be much more convenient than to do it directly in MATLAB. Notice, however, that there is some penalty paid in execution speed, because the computation is not vectorized and therefore a slower `for-loop` needs to be used instead.

<sup>11</sup>The demo files can be evaluated simply by setting the Lisp listener to package BC by form `(in-package :bc)` and then loading the demo file by `(load "mdemos/dsp/sinosc")`. A more flexible way is to open the file in editor that is linked to Lisp for interactive programming. In CLISP versions of BC3 this can be done using for example the Emacs editor and SLIME linking to Lisp, see BlockCompiler installation instructions.

The block `.sin-osc` can have two parameter inputs<sup>12</sup>, named `freq` and `amp1`. Here they are given values by *keyword syntax*, where a keyword name is written with a colon character (`:freq`, `:amp1`), and the value of each keyword is given thereafter. In this case, both parameters have a constant value.

The second line of the patch creates a named probe variable “out”, which is then connected (third line) to the sinewave output.

The patch can now be exported to MATLAB by executing the following script (in file “`mdemos/dsp/sinosc`”):

```
(matlab-response sinosc ;;; model export to Matlab
    :samples 441
    :outputs '("out")
    :post "
figure(10); clf; subplot(2,1,1);
plot_sig(out_resp,SRATE); grid on;
xlabel('Time [s]');
ylabel('Amplitude');")
```

which first specifies how many sample steps are simulated and then the names of probe output(s) to be observed. The string after keyword `:post` is MATLAB code that is exported as such for graphical presentation of the simulation results. `SRATE` is the sample rate for the patch (default 44100 Hz), and `out_resp` (variable name + `_resp`) is variable name in MATLAB that keeps the response collected from the probe variable `out`. After evaluating this export script, execution of `bc_resp` in MATLAB runs the whole simulation and shows the results as plotted in Fig. 11.1.

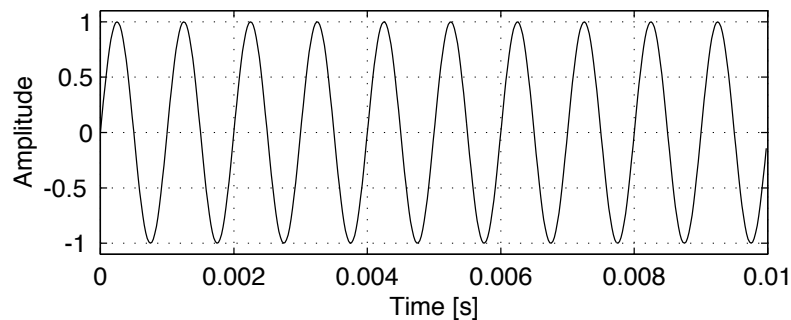


Figure 11.1: Output from MATLAB simulation of sinewave oscillator.

In the sine-wave simulation above, as with other patch models in this chapter, both model creation and export of patch for simulation in MATLAB can be done in BC3 by evaluating a form such as `(load "mdemos/dsp/sinosc")` in the Lisp listener window. For further details, see the BC3 instructions included in the installation package.

Notice also that simulation in MATLAB is done in a for-loop by calling the `bc:step` function, which is much slower than real-time simulation in BC itself, and may with complex models and a large number of sample steps become unpractically slow<sup>13</sup>.

<sup>12</sup>The difference of signal and parameter inputs is not so obvious here, however.

<sup>13</sup>Octave is found to be even slower in this respect.

### 11.2.2 Exporting simulation results to MATLAB/Octave

The previous case of sine-wave oscillator modeling can be done as well by doing the simulation in BlockCompiler and then exporting only the result to MATLAB for visualization. This can be done by script (in file "mdemos/dsp/sinosc2"):

```
(matlab-result sinosc ;;; simulation and result to Matlab
      :samples 441
      :output "out"
      :post "
figure(10); clf; subplot(2,1,1);
plot_sig(out_resp,SRATE); grid on;
xlabel('Time [s]');
ylabel('Amplitude');")
```

with equal results as above. This is typically faster than simulation in MATLAB because the stepping is done by compiled C-code on BC, and only the result is transferred to MATLAB.

### 11.2.3 Real-time simulation in BlockCompiler

A version of sine-wave oscillator for realtime sound output is done by script (in file "rtdemos/dsp/sine-tone"):

```
(defpatch sine-tone ((freq (.var 1000.0)) ;;; frequency
                    (ampl (.var 0.5)) ;;; amplitude
                    (sin (.sin-osc :freq freq :ampl ampl))
                    (out (.da))) ;;; D/A converter
(-> sin (inputs out))) ;;; sine wave to D/A
```

This simple patch makes an oscillator with constant frequency (1000 Hz) and amplitude (0.5, max level 1.0) and connects it to sound output (both left and right channel) of D/A converter block .da. Real-time streaming is started by evaluating (run-patch sine-tone), and the streaming is stopped by evaluating (stop-patch sine-tone).

Real-time simulation requires that the installation of BlockCompiler has included adding the gcc compiler and PortAudio sound I/O features.

### 11.2.4 Exporting a patch to Pd external

**This is currently non-functional (will be rewritten from the previous version of BlockCompiler).**

Pd (PureData) [82] is a block-based graphical programming environment for real-time audio signal processing. It is flexible and interactive for wiring patches with unidirectional data flow, but does not support bidirectional port-based connections used in physics-based modeling, and the buffer-based data transfer to speed up execution doesn't is not compatible to the philosophy of physical interactions. However, inside of a Pd block the internal connections can be of any form. Therefore it is possible to create physical submodels in BC and to export them to Pd as far as the external inputs and outputs are for unidirectional data flow only.

Sine-wave example to be added here ....

## 11.3 DSP models in BC for MATLAB

In this subsection we will study a couple of simple DSP models and how they are realized in BlockCompiler to be simulated in MATLAB. The main goal is to learn the basic syntax used in BC in order to get ready to move to physics-based modeling in the following section.

### 11.3.1 Signal distortion by smooth nonlinearity

Nonlinearities in signal processing are often highly undesirable, but sometimes they are useful or just what is needed. Nonlinear distortion is utilized for example in electric guitar amplifiers in order to make the otherwise ‘dry’ timbre of the guitar richer. Here we make a very simple distortion unit by feeding signals through a hyperbolic tangent function, which has a smoothly saturating input-output curve, as plotted in Fig. 11.2.

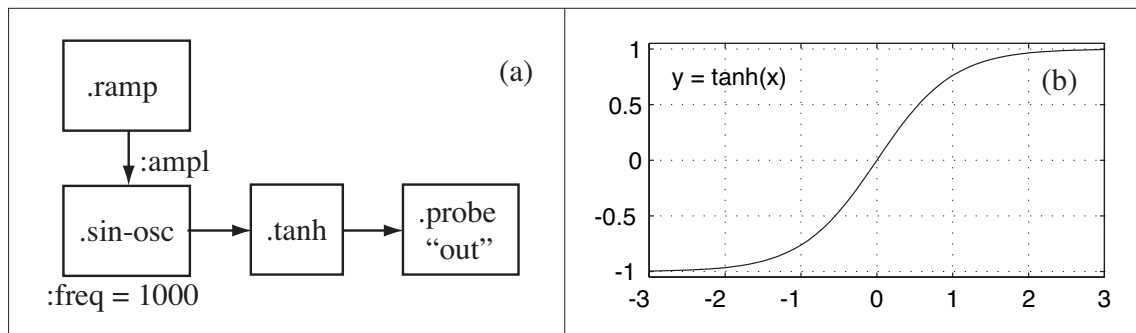


Figure 11.2: Smoothly saturating distortion by hyperbolic tangent function  $y = \tanh(x)$ . (a) BC patch diagram, (b) nonlinear characteristics of function  $\tanh$ .

The distortion generator is defined (in file “mdemos/dsp/nonlin2”) by patch:

```
(defpatch nonlin2 ((src (.ramp 150.0))
                  (sin (.sin-osc :freq 1000.0 :ampl src)))
  (-> sin (.tanh) (.probe "out")))
```

and the distorted signal is plotted in Fig. 11.3.

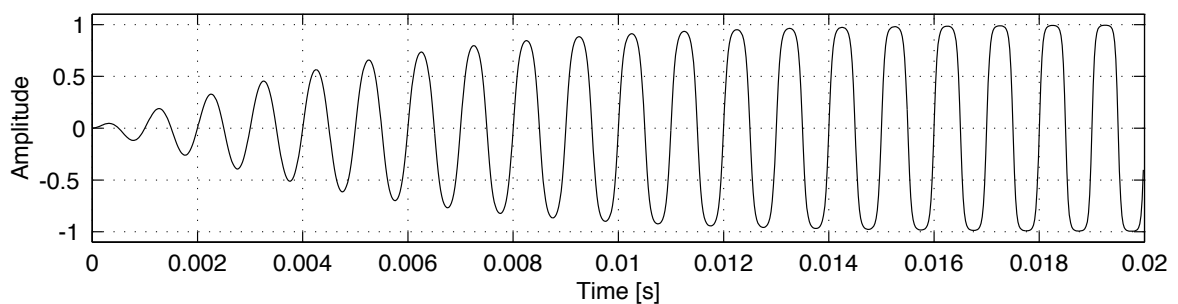


Figure 11.3: Sine waveform of increasing amplitude distorted by the hyperbolic tangent nonlinearity of Fig. 11.2.

In the patch, `nonlin2` the BC block `.ramp` generates a linearly increasing function of time with a slope of 150 per second, so that within a simulation period of 20 ms it grows from zero to 3. This ramp is used to control the amplitude of a sinewave oscillator (`.sin-osc`, line 2). Notice that the blocks `.tanh` and `.probe` are created inside the chaining function `->`, which is possible because they are referred to only once.

### 11.3.2 First-order lowpass filter

Digital filters are an important part of processing in BC. They will include feedback paths, so it is instructive to see the principle how they are dealt with in BlockCompiler. A first-order lowpass filter with unity gain at DC (zero frequency) has the  $z$ -transform function

$$H(z) = \frac{1 - k}{1 - kz^{-1}} \quad (11.1)$$

Figure 11.4 shows the diagram of the filter, constructed of elementary DSP blocks, for  $k = 0.995$ , which makes a lowpass cutoff frequency of  $f_{LP} \approx (1 - k)f_s/(2\pi) \approx 35$  Hz when the sample rate is  $f_s = 44100$  Hz.

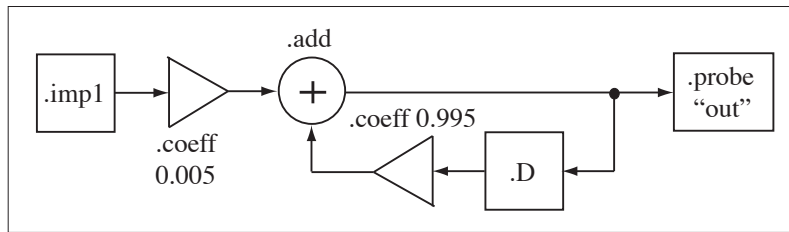


Figure 11.4: Diagram of first-order lowpass filter.

The filter has been implemented by the following BC code (in file “`mdemos/dsp/filters/LPF1`”):

```
(defpatch LPF1 ((adder (.add)) ;;; adder
                (k 0.995)) ;;; coefficient
  (-> (.imp1 1.0) (.coeff (- 1.0 k)) adder)
  (-> adder (.d) (.coeff k) (in adder 1))
  (-> adder (.probe "out")))
```

The first code line creates an adder (of two inputs), and the second line specifies the filter coefficient  $k = 0.995$ . Connecting the patch by `->` forms starts from a unit impulse generator through coefficient  $1 - k$  to the first (= default) input of the adder. The second wiring line connects the feedback loop through a unit delay and coefficient  $k$  back to the (second input of) adder. Finally the adder output is connected to the probe variable “out”.

Figure 11.5 plots the impulse response and magnitude response of the filter.

A slightly more efficient implementation of the first-order lowpass filter is available as the block `.lp1` in the BC3 DSP library.

### 11.3.3 Karplus-Strong string synthesis

One of the classical models of sound synthesis is the Karplus-Strong (K-S) algorithm [79, 80, 69], which has been found useful especially in synthesizing string instrument sounds. The

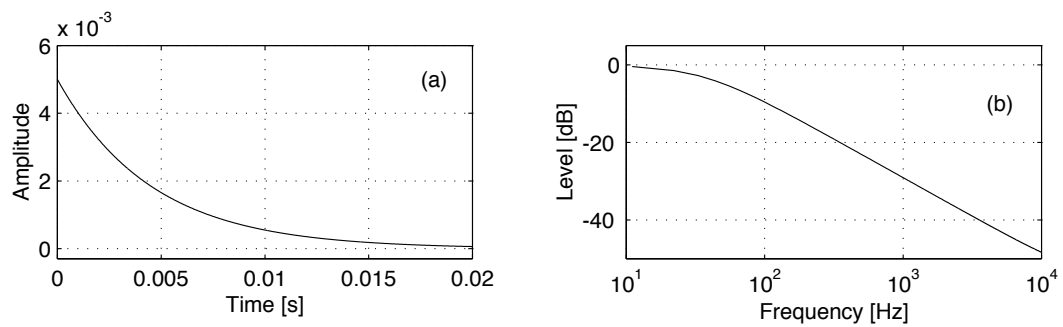


Figure 11.5: (a) Impulse response and (b) magnitude response of the lowpass filter.

following BC model is a simple implementation of it for exporting to MATLAB for response visualization and sound output. Figure 11.6 shows the principle of the K-S model, where an impulse `.imp1` is fed to a feedback loop consisting of a two-tap FIR filter and delay of 200 samples.

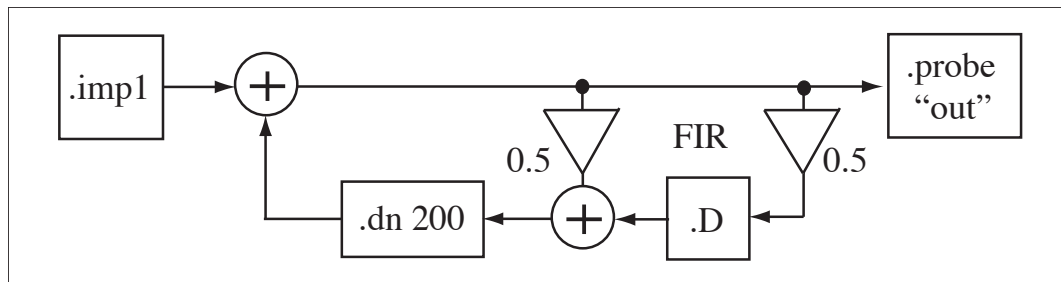


Figure 11.6: Simple Karplus-Strong string synthesis diagram.

The model is created by script (in file “`mdemos/dsp/instru/KS-simple`”). Notice how the whole diagram can be wired as a single chain of output-input connections.

```
(defpatch KS-simple ((add (.add)))
  (-> (.imp1) add (.fir :coeffs '(0.5 0.5))
      (.dn 200) (in add 1) (.probe "out")))
```

Figure 11.7 depicts the time-domain and the frequency-domain responses of the KS-model. The time-domain response consists of a series of impulses from the delay loop, each one slightly lowpass-filtered from the previous one. This means that high frequencies attenuate faster than the low-frequencies, which is typical to string vibration. The frequency response shows the same information as a series of harmonic peaks. The response sounds like an electric guitar string without sound effects.

## 11.4 Real-time DSP in BC

BlockCompiler is, in addition to model building, an efficient runtime engine for real-time simulation. As described in the beginning of this chapter, a model constructed in BC can generate C code and compile it for real-time execution. Availability of audio I/O makes it possible to record, process, and play sound. In this section we will explore a couple of simple DSP-based examples.

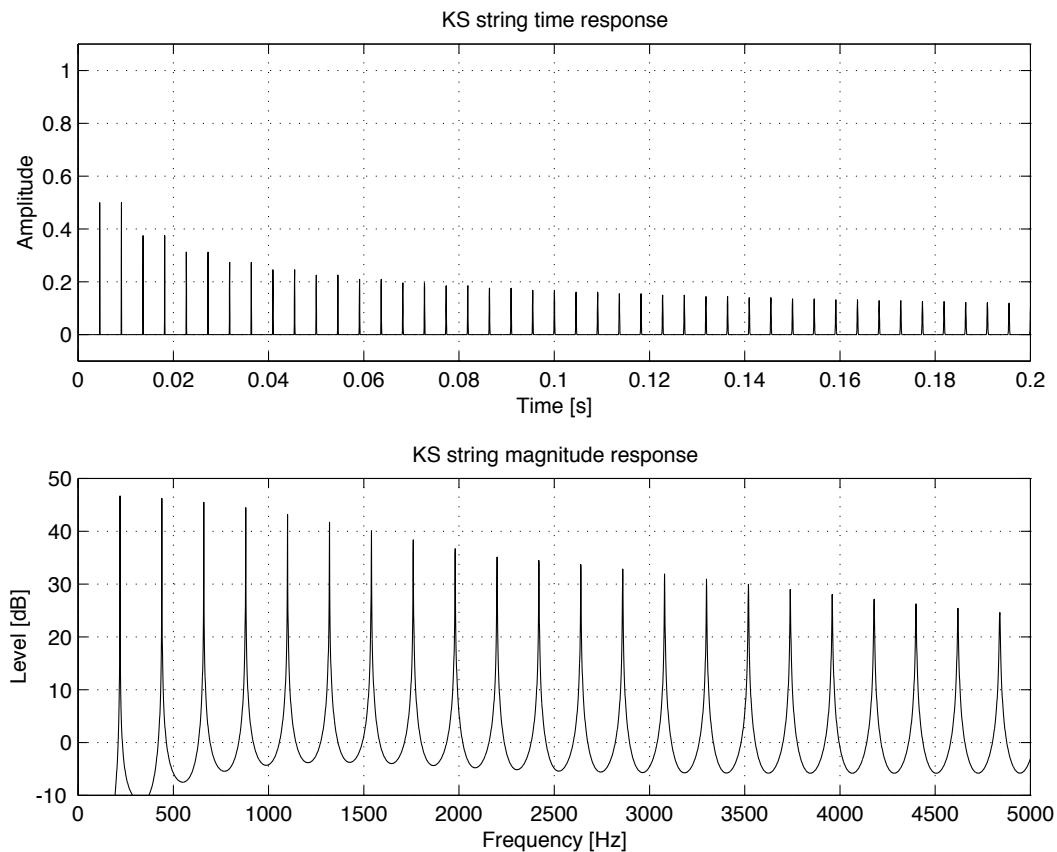


Figure 11.7: Karplus-Strong model response: (top) in time domain and (bottom) frequency domain.

### 11.4.1 Modulated sine-wave synthesis

The sine-wave synthesis in Section 11.2.3 can be enriched by modulating it by another sine wave. Frequency modulation synthesis (FM-synthesis) [77] is one of the most traditional sound synthesis techniques. Figure 11.8 shows the synthesizer patch diagram.

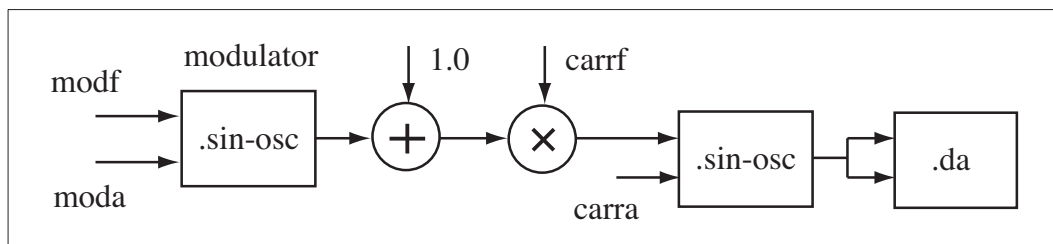


Figure 11.8: Patch diagram for a simple FM synthesizer.

The patch is realized by the following script (in file “rtdemos/dsp/sine-tone2”):

```
(defpatch fm ((modf (.var 300.0))      ;;; modulator frequency
              (moda (.var 0.95))      ;;; modulator amplitude
              (carrf (.var 500.0))    ;;; carrier frequency
              (carra (.var 0.50))     ;;; carrier amplitude
              (msin (.sin-osc :freq modf :ampl moda)))
```

```

(csin (.sin-osc :ampl carra))
(out (inputs (.da)))) ;;; D/A converter
(defun set-modf (f) (setf (at modf) f)) ;;; set mod freq
(defun set-moda (a) (setf (at moda) a)) ;;; set mod ampl
(defun set-carrf (f) (setf (at carrf) f)) ;;; set carr freq
(-> (.mul (.add msin 1.0) carrf) (param csin 'freq))
(-> csin out)) ;;; sine wave to D/A

```

This code first creates the control variables and sine-wave oscillators, doing also most of the parameter wiring. The `defun` forms create functions that can be called to set values for the parameters also when the patch is running. The two last lines do the rest of patch connections.

When the script has been evaluated, it can be started by `(run-patch fm)`. The default value of modulating frequency can be set by `set-modf` function, e.g., by `(set-modf 410.0)`. Modulation depth can be set by `set-moda` and carrier frequency by `set-carra` functions.

### 11.4.2 Echo processor

The next example is possible only if you have microphone input and loudspeaker output available for your computer in addition to sound input and sound output through PortAudio. The patch is a delay line to delay the signal from input to output. For more fun, you may modulate the delay to create kind of Doppler effect. A block diagram for the patch is shown in Fig. 11.9.

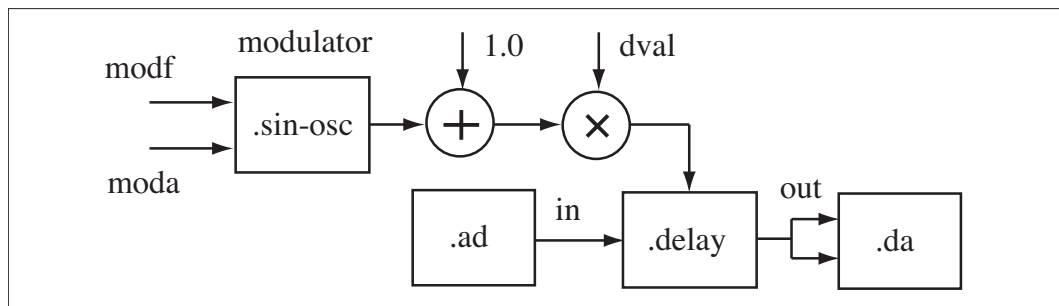


Figure 11.9: Simple delay with controllable delay and delay modulation.

The patch code is as follows (in file “mdemos/rtdemos/dsp/delay1”):

```

(defpatch delay ((modf (.var 4.0)) ;;; modulator frequency
(modf (.var 0.0)) ;;; modulator depth
(msin (.sin-osc :freq modf :ampl moda))
(dval (.var 0.1)) ;;; nominal delay
(dx (.mul (.add msin 1.0) dval)) ;;; delay
(dblock (.delay :time dx :max-time 0.2))
(in (.ad)) ;;; A/D converter
(out (.da)) ;;; D/A converter
(defun set-modf (f) (setf (at modf) f))
(defun set-moda (a) (setf (at moda) a))
(defun set-dval (x) (setf (at dval) x))
(-> in dblock (inputs out))) ;;; signal path

```



The delay modulator is implemented as the modulator for the FM synthesizer in the previous example. The initial value of the modulation depth is zero, i.e., no modulation. Modulation depth can be set between 0...1 by function `set-modu`. The delay block is a 3rd order Lagrange-interpolated delay-line with maximum delay of 0.2 seconds. Nominal delay `dval` can be set by function `set-dval`.

Notice that the acoustic path from loudspeaker to microphone may cause 'howling' if the gain of the loop is too high. Notice also that the overall signal delay from input to output includes the latencies due to sound driver software, so that it may be considerably longer than the nominal value set in the patch.

### 11.4.3 Real-time Karplus-Strong synthesis

A non-realtime string synthesis was described in 11.3.3. A simple real-time version of that model can be created by script (in file "rtdemos/instru/KS-rtsimple"):

```
(defparameter *KS-table*
  '(0.1 0.2 0.3 0.4 0.5 0.4 0.3 0.2 0.1))

(defpatch KS-realtime ((tr (.trig-data *KS-table*))
                      (adder (.add)))
  (defun pluck () (trig tr))
  (-> tr adder (.fir :coeffs '(0.498 0.498))
    (.delay :length 200) (in adder 1) (inputs (.da))))
```

The variable `*KS-table*` is first made to keep a simple triggerable wavetable with sawtooth-like waveform. The `KS-realtime` model is a single-delay loop patch, the output of which is connected to sound output. Function `pluck` is defined to trigger a pluck. When the patch is made running, then executing (`pluck`) excites the string model with new pluck every time the form is evaluated.

## 11.5 Physical modeling in BC

Physics-based modeling is different from DSP algorithms in that physical elements are connected through two-directionally interacting ports instead of one-directional data flow through terminals in DSP. The following two examples characterize how physics-based modeling is done in BC and exported to MATLAB for visualization.

### 11.5.1 RC circuit

The first example of physics-based modeling in BC is to compute how a capacitor is charged from a constant voltage source when the initial voltage of the capacitor at time  $t = 0$  is zero. Figure 11.10 depicts the electric circuit diagram and the block diagram using BC symbols for voltage source and capacitor. The BC patch for it (in file "mdemos/phys/ele/RC1") is:

```
(defpatch RC1 ((c (.C 2e-6))          ;;; C 2 uF
               (e (.E 1.0 1e3)))      ;;; 1V 1 kOhm
  (.par e c)                          ;;; parallel
  (-> (.voltage c) (.probe "CV"))      ;;; C voltage
  (-> (.current c) (.probe "CI")))     ;;; C current
```

The first line instantiates a WDF capacitor of  $2 \mu\text{F}$  and the second line makes a constant voltage source of 1 Volt and internal resistor of  $1 \text{ k}\Omega$ . Line 3 connects them in parallel. The last two lines apply physical probes `.voltage` and `.current` to the capacitor (actually to its port) to observe the voltage and the current, respectively, and connect them to output probe blocks.

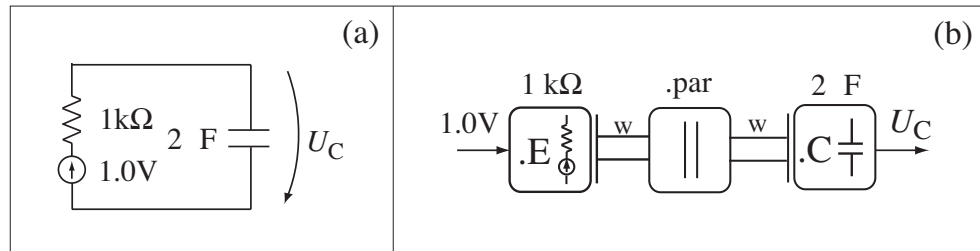


Figure 11.10: Capacitor charging from a constant voltage source: (a) RC circuit and (b) BC model diagram.

The results of RC circuit simulation (in MATLAB exported from BC) as step function responses are plotted in Fig. 11.11, where the voltage and the current of the capacitor are shown as functions of time. They are exponential curves, as expected from the circuit theory.

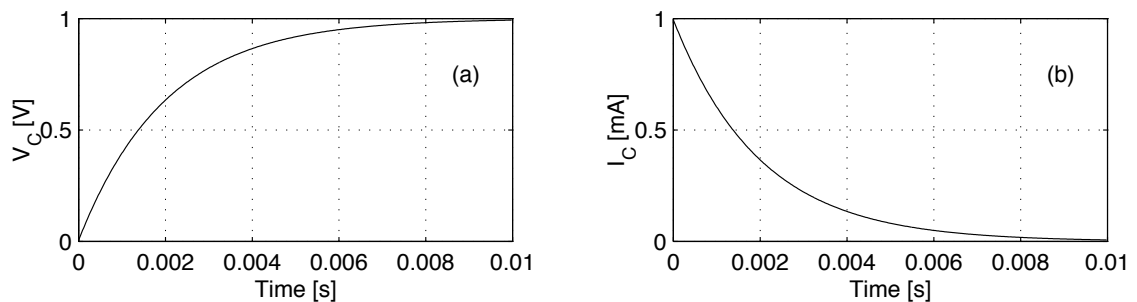


Figure 11.11: (a) Voltage and (b) current of the capacitor in the circuit of Fig. 11.10 when the voltage of the capacitor is zero at the beginning of simulation.

### 11.5.2 Delay line circuit

The next example illustrates a simple spatially distributed circuit consisting of a voltage source, a DWG delay line, and a termination resistance. This corresponds to a transmission line that is terminated with a non-perfectly matched impedances, which results in transient oscillation before reaching the final steady state for a step function excitation.

Figure 11.12 shows the electric circuit diagram and the block diagram using. The BC patch for it (in file “mdemos/phys/ele/DL1”) is:

```
(defpatch DL1 ((src (.E 1.0 0.1))    ;;; 1V, 0.1 Ohm
               (dl (.dline-n 10.0 :delay-length 10))
               (r (.R 100.0)))       ;;; 100 Ohm
  (.par src (port dl 0))             ;;; parallel
  (.par (port dl 1) r)               ;;; parallel
  (-> (.voltage r) (.probe "out"))) ;;; R voltage
```

The first line in code creates a voltage source with 1 V step function and  $0.1\ \Omega$  internal resistance. The second line makes a waveguide delay line of length 10 unit delays ( $0.227\text{ ms}$  at sample rate of  $44100\text{ Hz}$ )<sup>14</sup> and wave impedance of  $10\ \Omega$ . The third line creates a load termination resistance of  $100\ \Omega$ . The ports of the delay line are parallel connected to the source and the termination. Finally the voltage over the capacitor is probed as output signal.

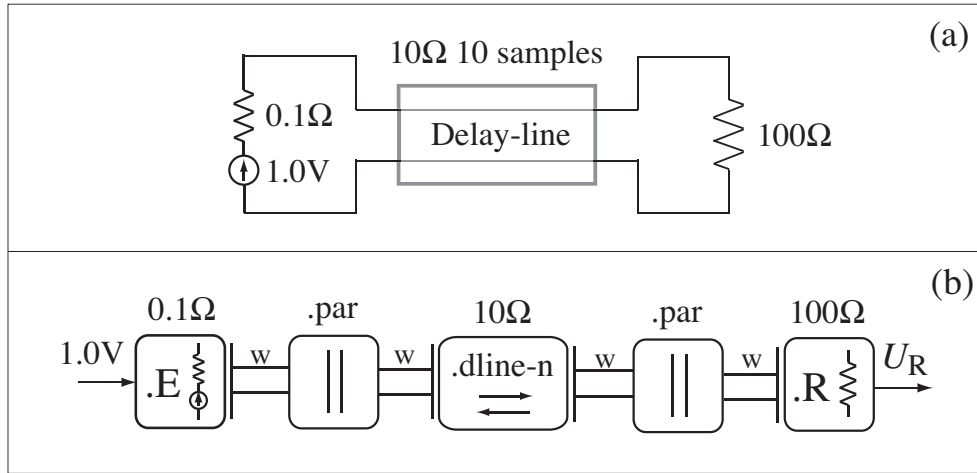


Figure 11.12: Delay line circuit with voltage step function as source and a resistor as termination: (a) circuit diagram and (b) BC model diagram.

Figure 11.13 plots the voltage step function response of the delay line circuit.

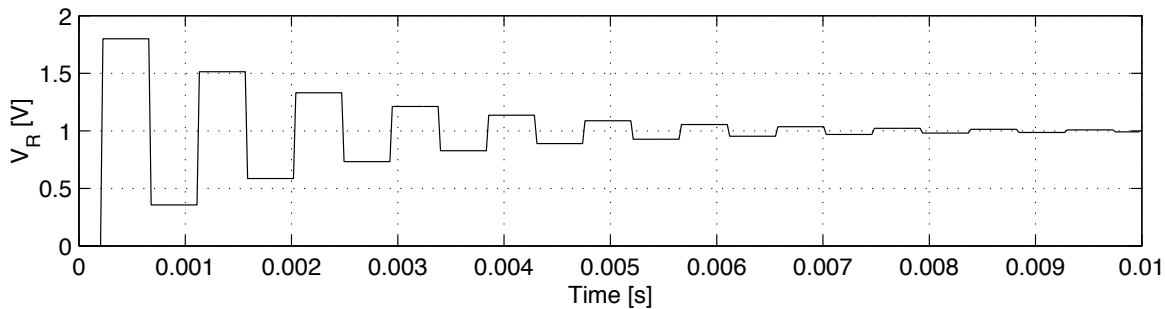


Figure 11.13: Voltage step function response of the circuit Fig. 11.12 measured as voltage over the termination resistance.

### 11.5.3 Delay line circuit made of unit delay lines

This example shows how the same delay-line behavior as above can be realized by composing the delay line using unit-length delay-line elements. This requires a bit more Lisp programming (in file “mdemos/phys/ele/DL1c”):

<sup>14</sup>In the electric domain this may correspond to a lossless transmission cable of length  $0.7 \cdot 300000\text{ km/s} \cdot 0.227\text{ ms} = 47670\text{ km}$  if the signal in the cable travels 70 % of the speed of light, which sounds quite unrealistic. It is however a realistic case for wave propagation in a mechanical or acoustical transmission line, for example a high-pitch sound in a wind instrument bore. For sound velocity of  $330\text{ m/s}$  in the air this corresponds to a tube of length about  $7.5\text{ cm}$ .

```

(defpatch DL1c ((z (.var 10.0))      ;;; wave Z
               (src (.E 1.0 0.1))    ;;; 1V, 0.1 Ohm
               (dx (.dline-1 z))     ;;; first delay
               (r (.R 100.0)))       ;;; 100 Ohm
  (.par (port dx 1) r)                ;;; parallel
  (loop for i from 1 below 10        ;;; indexing
        for di = (.dline-1 z)       ;;; new delay
        do (.pair (port dx 0) (port di 1)) ;;; pairwise
        do (setq dx di)              ;;; di -> dx
        finally
          (.par src (port di 0)))     ;;; parallel
  (-> (.voltage r) (.probe "out")))  ;;; voltage of R

```

In this script the difference to the previous one are the following. First a single unit delay line is made to `dx` and it is connected to the load resistor by `.par` for parallel connection of the two ports. Then the Lisp iteration form `loop` is used to create the rest (9) of the unit delay-line sections, one by one, and connecting to the previous section by `.pair`, which connects the ports directly together without an adapter. This is valid only when the port resistances are equal as it is here. The `setq` form sets the new section to local variable `dx` for the next iteration step. Finally the last unit delay-line section is connected to the voltage source.

The simulation of this circuit gives the same result as in Fig. 11.13. It is obvious that this version is not as efficient computationally than using the more optimized `.dline-n` element of the BlockCompiler library.

### 11.5.4 Definition and use of a macro block

The delay-line example above can be used further to show how new macro blocks are declared and used in BlockCompiler. The following script (in file “mdemos/phys/ele/DL1d”) defines a delay line named `.dline-nx` of controllable length (`delay-length`) and wave impedance (`z`), therefore being functionally equivalent with block `.dline-n` used before:

```

(def-macro-block .dline-nx (&key z delay-length)
  (let* ((d0 (.dline-1 z)))
    (loop with dx = d0
          for i from 1 below delay-length
          for di = (.dline-1 z)
          do (.pair (port di 1) (port dx 0))
          do (setq dx di)
          finally
            (set-ports (port dx 0) (port d0 1)))))

```

In this script `def-macro-block` specifies the name `.dline-nx` and keywords for instantiating the new type of delay-lines. The body of this definition macro is quite similar to the `loop` form used in the previous subsection. Here `dx` is a variable local to the `loop` form to keep the first created unit delay line. Variable `di` keeps the currently created delay line. The `set-ports` form collects the two ports to be made the ports of the macro block and visible to the user of it.

After evaluating the definition script the new delay-line element `.dline-nx` is ready for use. File “`mdemos/phys/ele/DL1d`” includes also a script to make the same delay-line network example as demonstrated before in Subsection 11.5.2. There are only two minor differences: block name and keyword syntax `:r` for specifying the wave impedance:

```
(defpatch DL1d ((src (.E 1.0 0.1))    ;;; 1V, 0.1 Ohm
               (dl (.dline-nx :r 10.0 :delay-length 10))
               (r (.R 100.0)))        ;;; 100 Ohm
  (.par src (port dl 0))               ;;; parallel
  (.par (port dl 1) r)                 ;;; parallel
  (-> (.voltage r) (.probe "out")))    ;;; R voltage
```

### 11.5.5 Simple nonlinearity: Rectification by diode

A simple case of nonlinearity is demonstrated in this subsection. The ideal diode, presented in Section 6.3.3, is now used to simulate rectification of a sinewave generator output. A capacitor is used to smooth the voltage that is supplied to a load resistor. The circuit diagram is shown in Fig. 11.14(a) and the BC model diagram in Fig. 11.14(b). The BC patch for the circuit (in file “`mdemos/phys/ele/Rect1`”) is:

```
(defpatch Rect1 ((src (.sin-osc :freq 50.0))
                 (e (.E src 20.0)) ;;; Ri = 20 Ohm
                 (d (.diode))
                 (r (.R 1.0e3)))
  (.root d (.ser e (.par r (.C 2.0e-4))))
  (-> (.voltage r) (.neg) (.probe "UR"))
  (-> (.current d) (.neg) (.probe "IE")))
```

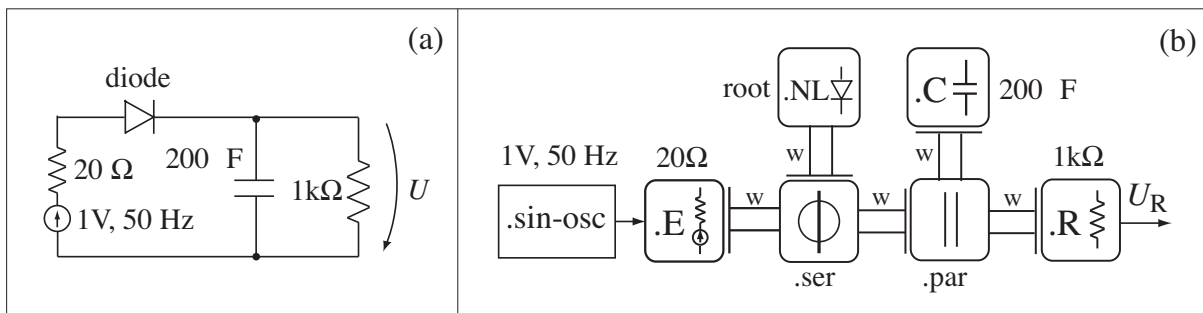


Figure 11.14: Rectification of sine wave source with an ideal diode, filter capacitor (1  $\mu$ F) and load resistor (1 k $\Omega$ ): (a) circuit diagram and (b) BC model diagram.

In the BC model script and Fig. 11.14(b) the load resistor (1 k $\Omega$ ) and filter capacitor (200  $\mu$ F) are first connected in parallel. This is connected in series with the sinewave voltage source, and this is finally connected to the nonlinear root element (diode). Notice the function `.root` that is used for the last connection. Remember that there can be only one nonlinearity as the root element of a standard WDF circuit (unless there are delay elements in between).

Notice also the `.neg` function to negate the voltage and current outputs. This is due to the sign conventions of the `.ser` adaptor connection to get the final results into more intuitive

form. Figure 11.15 plots the voltage over the load resistor and the current through the diode when the sinewave source starts at time moment zero.

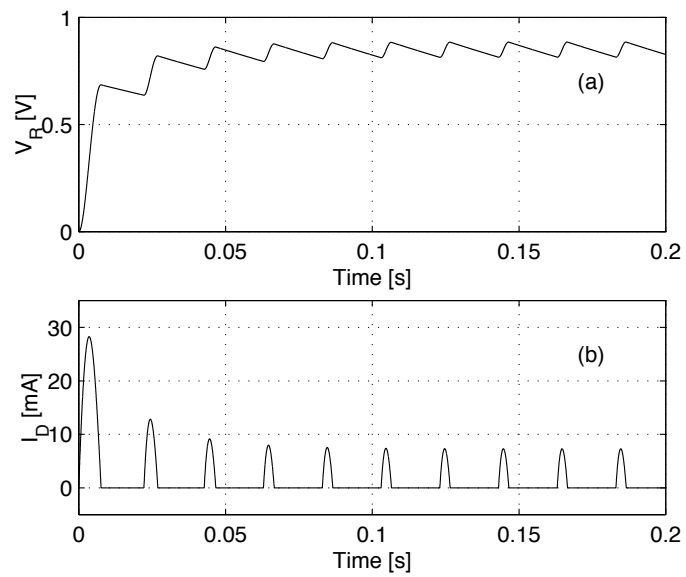


Figure 11.15: (a) Voltage over load resistor and (b) current through diode in the rectifier circuit of Fig. 11.14.



# Bibliography

- [1] “Gustav Kirchhoff.” URL: [http://en.wikipedia.org/wiki/Gustav\\_Kirchhoff](http://en.wikipedia.org/wiki/Gustav_Kirchhoff).
- [2] J. W. Nilsson and S. A. Riedel, *Electric Circuits*. Prentice-Hall, 6th ed., 1999.
- [3] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.
- [4] A. V. Oppenheim, A. S. Willsky, and S. H. Navab, *Signals and Systems*. Prentice-Hall, second edition ed., 1996.
- [5] L. Trautmann and R. Rabenstein, *Digital Sound Synthesis by Physical Modeling Using the Functional Transformation Method*. New York, NY: Kluwer Academic/Plenum Publishers, 2003.
- [6] T. I. Laakso, V. Välimäki, M. Karjalainen, and U. K. Laine, “Splitting the unit delay — Tools for fractional delay filter design,” *IEEE Signal Processing Magazine*, vol. 13, pp. 30–60, Jan. 1996.
- [7] I. N. Bronshtein, K. A. Semendyayev, G. Musiol, and H. Muehlig, *Handbook of Mathematics*. Berlin: Springer Verlag, 2004, 4th Edition.
- [8] K. F. Riley, M. B. Hobson, and S. J. Bence, *Mathematical Methods for Physics and Engineering*. Cambridge University Press, 2002, Second Edition.
- [9] Wai-Kai Chen (ed.), *The Circuits and Filters Handbook*. CRC Press, 2003.
- [10] “Spice software.” URL: <http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/>.
- [11] “Resistor.” URL: <http://en.wikipedia.org/wiki/Resistor>.
- [12] “Capacitor.” URL: <http://en.wikipedia.org/wiki/Capacitor>.
- [13] “Inductor.” URL: <http://en.wikipedia.org/wiki/Inductor>.
- [14] “Transformer.” URL: <http://en.wikipedia.org/wiki/Transformer>.
- [15] “Gyrator.” URL: <http://en.wikipedia.org/wiki/Gyrator>.
- [16] L. L. Beranek, *Acoustics*. American Institute of Physics, 1988, Third printing.
- [17] J. Borwick, *Loudspeaker and Headphone Handbook*. Focal Press, 2001, Third Edition.
- [18] M. Colloms, *High-Performance Loudspeakers*. Wiley, 2005, Sixth Edition.



- [19] A. V. Oppenheim, A. Willsky, and I. Young, *Signals and Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1983.
- [20] A. V. Oppenheim and R. W. Schaffer, *Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.
- [21] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.
- [22] S. Mitra and J. Kaiser, eds., *Handbook of Digital Signal Processing*. Wiley-Interscience, 1993.
- [23] J. Strawn, ed., *Digital Audio Signal Processing, An Anthology*. Willian Kaufmann Inc., 1985.
- [24] T. W. Parks and C. S. Burrus, *Digital Filter Design*. New York: John Wiley & Sons, Inc., 1987.
- [25] L. B. Jackson, *Digital Filters and Signal Processing*. Boston: Kluwer Academic Publ., 1989.
- [26] S. Haykin, *Modern Filters*. New York: Macmillan Publ. Co., 1989.
- [27] J. D. Markel and J. H. G. Gray, *Linear Prediction of Speech Signals*. Berlin: Springer-Verlag, 1976.
- [28] J. L. Kelly and C. C. Lochbaum, "Speech synthesis," in *Proc. Fourth Int. Congr. Acoustics*, (Copenhagen, Denmark), pp. 1–4, Sept. 1962.
- [29] H. W. Strube, "The meaning of the Kelly-Lochbaum acoustic-tube model," *J. Acoust. Soc. Am.*, vol. 108, pp. 1850–1855, October 2000.
- [30] C. Christopoulos, *The Transmission-Line Method TLM*. New York, NY: IEEE Press, 1995.
- [31] J. O. Smith, "A new approach to digital reverberation using closed waveguide networks," in *Proc. 1985 Int. Computer Music Conf.*, (Vancouver, Canada), pp. 47–53, 1985.
- [32] J. O. Smith, *Music Applications of Digital Waveguides*. Tech. Rep. STAN-M-39, CCRMA, Stanford University, CA, 1987.
- [33] J. O. Smith, "Physical modeling using digital waveguides," *Computer Music J.*, vol. 16, no. 4, pp. 74–91, 1992.
- [34] J. O. Smith, *Applications of Digital Signal Processing to Audio and Acoustics*, ch. Principles of Digital Waveguide Models of Musical Instruments, pp. 417–466. Kluwer Academic Publishers, Feb 1998.
- [35] J. O. Smith, *Physical Audio Signal Processing: Digital Waveguide Modeling of Musical Instruments and Audio Effects*. 2004. August 2006 Edition, <http://ccrma.stanford.edu/~jos/pasp/>.
- [36] N. H. Fletcher and T. D. Rossing, *The Physics of Musical Instruments*. Springer-Verlag, New York, 1991.

- [37] C. Valette, "The mechanics of vibrating strings," in *Mechanics of Musical Instruments* (A. Hirschberg, J. Kergomard, and G. Weinreich, eds.), pp. 115–183, Berlin: Springer-Verlag, 1995.
- [38] V. Välimäki, *Fractional Delay Waveguide Modeling of Acoustic Tubes*. PhD thesis, 1994. Report 34, Helsinki University of Technology, Laboratory of Acoustics and Audio Signal Processing.
- [39] V. Välimäki, M. Karjalainen, and T. Kuisma, "Articulatory control of a vocal tract model based on fractional delay waveguide filters," in *1994 Int. Symp. Speech, Image Processing and Neural Networks (ISSIPNN'94)*, (Hong Kong), pp. 571–574, 1994.
- [40] T. Kuisma, "Computational modeling of the vocal tract and speech synthesis by using fractional delay filters," 1994. MSc Thesis, Helsinki University of Technology, Laboratory of Acoustics and Audio Signal Processing.
- [41] S. A. Van Duyne and J. O. Smith, "Physical modeling with the 2-D digital waveguide mesh," in *Proc. Int. Computer Music Conf.*, (Tokyo, Japan), pp. 40–47, 1993.
- [42] M.-L. Aird, J. Laird, and J. ffitch, "Modelling a drum by interfacing 2-D and 3-D waveguide meshes," in *Proc. Int. Computer Music Conf.*, (Berlin, Germany), pp. 82–85, Aug. 2000.
- [43] F. Fontana and D. Rocchesso, "Physical modeling of membranes for percussion instruments," *Acustica United with Acta Acustica*, vol. 84, no. 3, pp. 529–542, 1998.
- [44] L. Savioja, T. J. Rinne, and T. Takala, "Simulation of room acoustics with a 3-D finite difference mesh," in *Proc. Int. Computer Music Conf.*, (Aarhus, Denmark), pp. 463–466, Sept. 1994.
- [45] L. Savioja, *Modeling Techniques for Virtual Acoustics*. Helsinki University of Technology, 1999. Doctoral Thesis, Report TML-A3.
- [46] P. Huang, S. Serafin, and J. O. Smith, "A 3-D waveguide mesh model of high-frequency violin body resonances," in *Proc. Int. Computer Music Conf.*, (Berlin, Germany), pp. 86–89, Aug. 2000.
- [47] L. Savioja and V. Välimäki, "Interpolated rectangular 3-D digital waveguide mesh algorithms with frequency warping," *IEEE Trans. Speech and Audio Processing*, vol. 11, pp. 783–790, Nov. 2003.
- [48] D. T. Murphy, C. J. C. Newton, and D. M. Howard, "Digital waveguide mesh modelling of room acoustics: Surround-sound, boundaries and plugin implementation," in *Proc. Int. Conf. Digital Audio Effects (DAFx)*, (Limerick, Ireland), 2001.
- [49] M. J. Beeson and D. T. Murphy, "RoomWeaver: A digital waveguide mesh based room acoustics research tool," in *Proc. COST G6 Conf. Digital Audio Effects*, (Naples, Italy), pp. 268–273, Oct. 2004.
- [50] L. Savioja and V. Välimäki, "Reducing the dispersion error in the digital waveguide mesh using interpolation and frequency-warping techniques," *IEEE Trans. Speech and Audio Processing*, vol. 8, pp. 184–194, Mar. 2000.

- [51] S. Van Duyne and J. O. Smith, "The tetrahedral digital waveguide mesh," in *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, (New Paltz, NY), pp. 463–466, 1995.
- [52] L. Savioja and V. Välimäki, "Multiwarping for enhancing the frequency accuracy of digital waveguide mesh simulations," *IEEE Signal Processing Letters*, vol. 8, pp. 134–136, May 2001.
- [53] F. Fontana and D. Rocchesso, "Online correction of dispersion error in 2D waveguide meshes," in *Proc. Int. Computer Music Conf.*, (Berlin, Germany), pp. 78–81, Aug. 2000.
- [54] S. Stoffels, "Full mesh warping techniques," in *Proc. COST G-6 Conf. on Digital Audio Effects*, (Verona, Italy), pp. 223–228, Dec. 2000.
- [55] F. Fontana, "Computation of linear filter networks containing delay-free loops, with an application to the waveguide mesh," *IEEE Trans. Speech and Audio Processing*, vol. 11, no. 6, pp. 774–782, 2003.
- [56] J. Strikverda, *Finite Difference Schemes and Partial Differential Equations*. Grove, Ca: Wadsworth and Brooks, 1989.
- [57] S. Bilbao, *Wave and Scattering Methods for Numerical Simulation*. John Wiley and Sons, 2004. ISBN: 0-470-87017-6.
- [58] A. Fettweis, "Wave digital filters: Theory and practice," *Proc. IEEE*, vol. 74, pp. 270–327, Feb. 1986.
- [59] S. S. Lawson and A. R. Mirzai, *Wave Digital Filters*. New York: Ellis Horwood, 1990.
- [60] A. Sarti and G. De Poli, "Toward nonlinear wave digital filters," *IEEE Trans. Signal Processing*, vol. 47, pp. 1654–1668, June 1999.
- [61] G. De Sanctis, A. Sarti, and S. Tubaro, "Automatic synthesis strategies for object-based dynamical physical models in musical acoustics," in *6th Int. Conf. Digital Audio Effects (DAFx-03)*, (London), 2003.
- [62] R. Rabenstein and S. Petrausch, "Block-based physical modeling," in *5th Vienna Symposium on Mathematical Modelling (MATHMOD)*, (Vienna), pp. 2.1–2.17, 2006.
- [63] D. Fränken, K. Meerkötter, and J. Waßmuth, "Passive parametric modeling of dynamic loudspeakers," *IEEE Transactions on Speech and Audio Processing*, vol. 8, pp. 885–891, Nov. 2001.
- [64] A. Fettweis, "Kirchhoff circuits, relativity, and beyond," in *European Conf. on Circuit Theory and Design (ECCTD 05)*, (Cork, Ireland), 2005. URL: [http://ecctd05.ucc.ie/public\\_presentation/449.Alfred\\_Fettweis.pdf](http://ecctd05.ucc.ie/public_presentation/449.Alfred_Fettweis.pdf).
- [65] A. Fettweis and K. Meerkötter, "On adaptors for wave digital filters," *IEEE Trans. Acoust., Speech and Sig. Proc.*, vol. ASSP-23, pp. 516–525, Dec. 1975.
- [66] S. Bilbao, "Robust physical modeling sound synthesis for nonlinear systems — Achieving musical sound synthesis through numeric simulation," *IEEE Signal Processing Magazine*, vol. 24, pp. 32–41, March 2007.

- [67] K. Meerkötter and R. Scholz, "Digital simulation of nonlinear circuits by wave digital filters," in *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, pp. 720–723, 1989.
- [68] G. Kubin, "Wave digital filters: Voltage, current, or power waves," in *Proc. 1985 IEEE Int. Conf. Acoust. Speech and Sig. Proc. (ICASSP-85)*, (Tampa, Florida), pp. 69–72, vol. 3, 1985.
- [69] M. Karjalainen, V. Välimäki, and T. Tolonen, "Plucked-string models: From the Karplus-Strong algorithm to digital waveguides and beyond," *Computer Music Journal.*, vol. 22, no. 3, pp. 17–32, 1998.
- [70] J. O. Smith, "Efficient synthesis of stringed musical instruments," in *Proc. Int. Computer Music Conf.*, (Tokyo, Japan), pp. 64–71, 1993.
- [71] M. Karjalainen, V. Välimäki, and Z. Jánosy, "Towards high-quality sound synthesis of the guitar and string instruments," in *Proc. Int. Computer Music Conf.*, (Tokyo, Japan), pp. 56–63, 1993.
- [72] J. Woodhouse, "Plucked guitar transients: Comparison of measurements and synthesis," *Acta Acustica united with Acustica*, vol. 90, no. 5, pp. 945–965, 2004.
- [73] M. Karjalainen, P. A. A. Esquef, P. Antsalo, A. Mäkitvirta, and V. Välimäki, "Frequency-zooming ARMA modeling of resonant and reverberant systems," *J. Audio Eng. Soc.*, vol. 50, no. 12, pp. 953–967, 2002.
- [74] S. A. Van Duyne, J. R. Pierce, and J. O. Smith, "Traveling-wave implementation of a lossless mode-coupling filter and the wave digital hammer," in *Proc. Int. Computer Music Conf.*, (Aarhus, Denmark), pp. 411–418, 1994.
- [75] M. van Walstijn, *Discrete-Time Modelling of Brass and Woodwind Instruments with Application to Musical Sound Synthesis*. PhD thesis, Univ. Edinburgh, Faculty of Music, Edinburgh, UK, Feb. 2002.
- [76] M. van Walstijn and M. Campbell, "Discrete-time modeling of woodwind instrument bores using wave variables," *J. Acoust. Soc. Am.*, vol. 113, pp. 575–585, Jan. 2003.
- [77] J. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the Audio Engineering Society*, pp. 526–534, 1973.
- [78] A. Härmä, M. Karjalainen, L. Savioja, V. Välimäki, U. K. Laine, and J. Huopaniemi, "Frequency-warped signal processing for audio applications," *J. Audio Eng. Soc.*, vol. 48, no. 11, pp. 1011–1031, 2000.
- [79] K. Karplus and A. Strong, "Digital synthesis of plucked-string and drum timbres," *Computer Music Journal.*, vol. 7, no. 2, pp. 43–55, 1983. Also published in Roads C. (ed). 1989. *The Music Machine*. The MIT Press. Cambridge, Massachusetts.
- [80] D. A. Jaffe and J. O. Smith, "Extensions of the Karplus-Strong plucked-string algorithm," *Computer Music J.*, vol. 7, no. 2, pp. 56–69, 1983. Also published in Roads C. (ed). 1989. *The Music Machine*, pp. 481–494. The MIT Press. Cambridge, Massachusetts.
- [81] "Simulink, software by MathWorks." URL: <http://www.mathworks.com>.

- [82] “Pd, software by Miller Puckett.” URL: <http://crca.ucsd.edu/~msp/software.html>.
- [83] “Max/msp, software by Cycling ’74.” URL: <http://www.cycling74.com/products/maxmsp.html>.
- [84] “Labview, software by National Instruments Co.” URL: <http://www.ni.com/>.
- [85] “Mustajuuri software.” URL: <http://www.tml.tkk.fi/~tilmonen/mustajuuri/>.
- [86] “PatchWork software.” URL: <http://www2.siba.fi/soundingscore/PWHomePage/patchwork.html>.
- [87] G. L. Steele, *Common Lisp, The Language*. Digital Press, 2nd edition, 1990.

# Appendix A

## BC3 Block Library User's Reference

This Appendix describes the basic block objects available in BlockCompiler, version BC3. The syntax and usage is characterized briefly for each block type and function to be used for patch scripting and programming.

### A.1 Manipulation of BlockCompiler patches and objects

This section describes the most important software constructs used to create patches and to inspect, compile, load, run, stop, control their parameters within the Lisp environment, and to export them to other execution platforms.

#### A.1.1 Patch creation

As described in Chapter 11, a patch can be created by using the special form `defpatch`. As a simple example

```
(defpatch px ((x1 (.var 1.2))      ;;; BC-variable in x1
              (x2 (.const 2.3))   ;;; BC-variable in x2
              (add (.add)))        ;;; BC-adder in add
  (-> x1 add)                      ;;; x1 to first input of add
  (-> x2 (in add 1))               ;;; x2 to second input of add
  (-> add (.probe "out")))        ;;; Probe output to "out"
```

When this for is evaluated, BC-variable made by `(.var 1.2)` is bound to Lisp (local) variable `x1`, BC-constant from `(.const 2.3)` is bound to Lisp variable `x2`, and a BC-block adder block from `(.add)` is bound to Lisp variable `add`. These sublists following `defpatch` and patchname `px` behave just like local variable definitions in `let*` Lisp form. Next the variable in `x1` and the constant in `x2` are wired to the two inputs of the adder, and finally the output of the adder is wired to a output object, a signal probe named "out" as created by form `(.probe "out")`. As the result of evaluating the entire form, `px` will be the Lisp (global) variable that keeps the patch object created.

If re-evaluating such a form, the previous `px`-named patch will be replaced by a new one. In the following, `px` is a generic name to refer to a patch to be manipulated.

### A.1.2 Patch structure

Lisp provides two functions for inspecting the software constructs within it:

- `(describe px)` prints the contents/properties of `px`
- `(inspect px)` opens an interactive inspector window in some Lisp environments, which is very convenient. In many simple Lisp environments, however, there is more limited support for `inspect`.

#### Internal structure of a patch

Patches (instances of class `patch`) are containers of complete computational models for physics-based modeling or signal processing. Patches are CLOS (Common Lisp Object System) objects with slots (instance variables) for storing the internal state and methods applicable for manipulating the objects. A short description of the internal object structure (main instance variables and access methods) of a patch is:

- `block-items`, accessed by `(block-items px)`, is a list of top-level blocks included in patch `px`. Macro-blocks in this list have further internal structure of `block-items`, etc.
- `lin-schedule`, accessed by `(lin-schedule px)`, is a linearized list of basic blocks included in the hierarchical structure of `px`, scheduled into a computable order.
- `variables`, accessed by `(variables px)`, is a list of variables used inside patch `px` for computation in C-code or Matlab m-files. These variables are most typically block outputs, but may be as well other variables inside blocks. For using these variables for inspection or runtime control, see Subsection A.1.4.
- `state`, accessed by `(state px)`, describes the state of a patch in terms of `NIL` = not compiled, loaded, or running; `:compiled` = patch is compiled but not loaded or running; `:loaded` = patch is compiled and loaded but not running; and `:running` = patch is compiled, loaded and running.
- `name`, accessed by `name px`, holds the symbol name of the patch. By default the name is generated by the system to be a unique indicator, but for specific purposes it can be specified by form, e.g. `(name 'my-block)`, in the local variable list of a `defpatch` definition.
- `srate`, accessed by `(srate px)`, keeps the main sample rate used for the patch. The sample rate can be specified in `defpatch`-expression, e.g., by `(srate 48000.0)` in the local variable part of patch definition. The platform should be able to support the given sample rate in order to run the patch. The default value of `srate` is 44100.
- `runtime`, accessed by `(runtime px)`, keeps a (platform-dependent) object that describe various things needed for runtime execution support. This is available after patch `px` has been compiled and loaded.

For further details, see the definition of patch class in the source code.

### Internal structure of a block

Block objects are constituents of patches, each one realizing a elementary or a composite definition of computation, i.e., a basic block or a macro block. Some of the most important instance variables common to each block (generic notation below is `bx` are:

- `inputs`, accessed by `(inputs bx)`, are signal rate synchronous inputs to block `bx`. Each input must be connected to one and only one output of another block. A single input can be accessed by index number of form `(in bx index)` or by input name, `(in bx name)`, if the inputs are named.
- `outputs`, accessed by `(outputs bx)`, are signal outputs from block `bx`. Each output can be used to any number of inputs or params of a block or it may be left open. A single output can be accessed in a way similar to accessing an input.
- `params`, accessed by `(params bx)`, are asynchronous parametric inputs that are intended for feeding values that are not necessarily updated for each signal sample. Each param input must be connected to one and only one output of another block. A single param input can be accessed in a way similar to accessing an input.
- `ports`, accessed by `(ports bx)`, are ports used to connect two-way physical wave-ports in blocks that support physical interaction. A single port input can be accessed in a way similar to accessing inputs and outputs.
- `host-patch`, accessed by `(host-patch bx)`, is a reference to a macro-block or patch, in which `bx` is included in the `block-items` list. The top-level `patch` points to `NIL`, i.e., it does not have a `host-patch`.
- `name`, accessed by `(name bx)`, is non-`NIL` if and only if the block is given a name when it is created.

For other features, see the definitions of the related block classes in the source code.

### A.1.3 Compilation and runtime operations of patches

When a patch has been created as described above, it can be manipulated in many ways. The most important functions/methods for this are:

- `(c-code px)` generates and prints to Lisp Listener C-code that corresponds to a single sample step of patch computation. Normally this happens inside patch compilation that writes the code into a file and compiles it, but calling `c-code` explicitly can be a useful debugging tool in BC programming.
- `(compile-patch px)` generates C-code for the patch and compiles it using a C-compiler. This can be done only in BC-versions that have run-time support.
- `(load-patch px)` both compiles and loads a run-time executable of patch `px`, initializing it ready for execution. This can be done only in BC-versions that have run-time support.
- `(step-patch px [steps] [report])` steps the patch computation by calling the runtime executable that must be loaded already. Here `steps` and `report` are optional: `(step-patch px)` just steps once. If optional argument `steps` is given as a positive integer, the patch will be stepped that many times. If both `steps` is given and `report` is `t`, the number of steps is executed and the execution time is reported (see also



function `step-patch-n`). Function `step-patch` can be used only in BC-versions that have run-time support.

- `(step-patch-n px [steps] [report])` is similar to `step-patch` except that all step loops are executed within the executable, thus avoiding repeated calls from Lisp for each step. Therefore the timing report better indicates the runtime behavior. Notice that Lisp break function does not stop this loop, so you cannot stop it before the loop is done. Function `step-patch-n` can be used only in BC-versions that have run-time support.
- `(run-patch px)` starts realtime execution of patch `px`. This requires BC runtime support, of course. Timing of patch steps is according to the audio driver, so this requires runtime support. The patch can be inspected and controlled in Lisp while it is running. Running of a patch doesn't require any AD- or DA-converter blocks in the patch, but of course practical audio patches in most cases use them. Several patches can be running simultaneously, in which case the DA-outputs will be summed and AD-inputs shared by the patches.
- `(stop-patch px)` stops realtime execution of patch `px`. Execution can be continued from the stopping state by calling `(run-patch px)` again.
- `(remove-patch px)` removes patch `px` from the list of valid patches on global variable `*patches*`. Normally the user doesn't have to take care of removing patches, and the patch list will be empty when starting BC again. Using `(defpatch px ...)` automatically replaces an existing patch with the same name. BC keeps a list of defined patches in variable `*patches*`.

### A.1.4 Controlling of a running patch

A patch that has been created (evaluated) is initialized so that its state, i.e., block outputs and variables, have their initial values in Lisp code. Exporting the patch (see A.2) will export the initial state as well. When the block is compiled and loaded, the variables are allocated in the runtime environment. These runtime variables can be accessed anytime from Lisp environment as far as the variables are declared accessible. For some block classes this is default behavior, for others only by declaration. If reading or writing a runtime variable is declared, reading or wiring is allowed also when the patch is running, thus allowing to control patch parameters at runtime. Blocks that allow this by default are: `.var` for reading and writing and `.probe` for reading. Access<sup>1</sup> can be done by forms

- Reading: `(at bx)` for scalars, `(at bx index)` for vectors, and `(at bx ind1 ind2)` for matrices, where `bx` is a proper block.
- Writing: `(setf (at bx) value)` for scalars, `(setf (at bx index) value)` for vectors, and `(setf (at bx ind1 ind2) value)` for matrices, where `bx` is a proper block and `value` is the value to be written.

Access to other runtime data objects is by default not enabled. To make such access possible, block outputs or internal variables of blocks can be made runtime readable or writable using

---

<sup>1</sup> Notice that these accesses to vectors or arrays take place elementwise, which means that the integrity of such a unit between runtime and Lisp environments is not guaranteed. To assure the integrity, buffering of such data structures must be done in the runtime environment for data synchronization, see ?.

forms such as `(setf (creader (out bx)) t)` for readability and `(setf (cwriter (out bx)) t)` for writability for the output of block `bx`. This must be done before the block is compiled. In this case the output can be read by a form such as `(at (out bx))`, or simply by `(at bx)`.

A special type of parametric control is triggering of an event. TO BE DOCUMENTED !!!

Add example!

## A.2 Export of BC patches

To be documented

- Exporting to Matlab by using `to-matlab`, `matlab-response` `matlab-result`, see examples.

- Exporting to `pd` (not ready)

- Exporting to ???

## A.3 Block classes, functions, and data structures

BlockCompiler is an object-oriented software environment, where computational blocks are objects (instances) of block classes. In simple use of BC the user does not necessarily deal with block classes, rather he/she deals with block creation functions and their parameters, as well as basic Lisp language constructs. In more advanced programming the use and definition of block classes is important.

Although the user of BlockCompiler doesn't have to know the details of BC implementation, it is conceptually important to understand the difference between objects in the model building environment ("Lisp environment") and runtime environment ("C-environment" or export environment). The Lisp environment allows for all possible constructs of that language, while the runtime environment objects are much more restricted in form and usage. To help keeping these two environments conceptually separate to the user, naming starting with dot, such as `.add`, refers to the objects for the runtime environments, and other symbols are for model building, not realized directly in the runtime patches.

Often a block creation function has the same name as the block class name, such as function expression `(.add)` that creates an adder of type `.add`, but the names are not necessarily equal. In the following the rules of using different parameter types in block creation are introduced.

### A.3.1 Parameters of block creation functions

In the example above the adder was created simply by calling `(.add)`. In a general case, the block creation function in the expression is followed by parameter values to specify details for the block. In Lisp syntax there are three basic types of parameters for functions: *required*, *optional*, and *keyword* parameters:

- **Required parameters** are obligatory, and typically they follow immediately after the function symbol. It is an error not to give a value for such an parameter. In the syntax definitions below the required parameters are symbols without brackets around them.
- **Optional parameters** can be given but are not necessary. Typically they follow the required parameters. There is normally a default value or behavior for the case an optional

parameter value is not given. Below the optional parameters are specified by symbols in brackets.

- **Keyword parameters** are typically optional but may also be required. They are given as pairs of a keyword and its value, e.g., ( `.add :out-type ' .long` ) makes an adder (with two inputs) and output data type of `.long`. Keyword symbols start with colon (:). In BC function forms the keyword parameters are used extensively, because they can be given in arbitrary order, while required and optional ones should be given in the order they are specified in the function definition. In block creation functions the keyword parameters are divided into two groups: generic and specific keyword parameters. The set of generic ones is simply denoted by [ `&keys` ] because they are common for wider sets of block classes. The specific keywords are more class-specific and need explanation separately for each block type.

### A.3.2 Generic keywords

The following keyword parameters are common to many block classes and are therefore described here:

- **:inputs**  
is used to specify the number of inputs for blocks that may have a variable number of inputs<sup>2</sup>, such as `.add`.
- **:name**  
can be used to give a name (symbol or string) to a block. This is useful when there is need to inspect the internal block structure of a patch or to search for a block inside a patch using function form ( `find-block patch block-name` ).
- **:out-type**  
is used to specify the output data type of most block classes. If this is not given the global default given in variable `*default-data-type*` is used, or a class-specific data type is used<sup>3</sup>. Data-types that are supported presently in most cases are: `.short`, `.long`, `.float`, and `.double`.
- **:mrate**  
is used in multirate (up- or down-sampled) computation. The value needs to be an integer greater than one for oversampled computation. This means that with keyword option `:mrate n`, e.g. `:mrate 10`, the operation of the block is carried out  $n$  times for each sample step of the patch where the block belongs to. For down-sampled computation the value of the keyword is rational number  $1/n$ , e.g. `:mrate 1/10`. Then the block operation is computed only for every 10th sample period of the host block.
- **:mphase**  
For polyphase multirate computation the initial phase (number between 0 and oversampling rate) is often needed and can be given by the keyword `:mphase`.

<sup>2</sup>In advanced programming the input objects can also be given as a list instead of the number of inputs.

<sup>3</sup>The data types of possible variables inside the block computation may follow the out-type, it may be specific to the block, or there may be separate keywords to control their types.

### A.3.3 Runtime data types and objects in BlockCompiler

A BlockCompiler patch consist of blocks and their interconnections. The following sections will document the block types available and how they are used as constituents of patches. This subsection will describe data typing conventions and practices used in BC.

#### Data types

The outputs, inputs, and param inputs, as well as possible internal variables of blocks are typed and sized. The following types are supported presently in most blocks:

- `.short` ~ short in C
- `.long` ~ long in C
- `.float` ~ float in C
- `.double` ~ double in C

The default type used in variables is controlled globally by variable `*default-type*` and it is set to `double` ....

Logic values,  
Complex ?

#### Data sizes

Scalar, vector, matrix

#### Data blocks

-

- Variable (`.var data [name]`)
- Constant (`.const data [name]`)
- Probe (`.probe name`)
- ref to: `.ad .da, .trig, .trig-data ???`

## A.4 Math functions and operations

### A.4.1 Multi-input arithmetic blocks

Arithmetic operations, such as `.add`, `.sub`, `.mul`, and `.div`, are blocks that can have any number of inputs, no param inputs, and one output. Their main features are described in Table A.1. In the simplest case, for example the form `(.add)` creates an adder block with two inputs and one output.

Table A.1: Multi-input arithmetic operation blocks:  $y$  = output,  $x$  = input.

Block name and syntax	Function	Description
<code>(.add [inputs] [&amp;keys])</code>	$y = \sum_{i=0}^{N-1} x_i$	sum up input values
<code>(.div [inputs] [&amp;keys])</code>	$y = x_0 / \prod_{i=1}^{N-1} x_i$	divide input values
<code>(.max [inputs] [&amp;keys])</code>	$y = \max_{i=0}^{N-1} x_i$	max of input values
<code>(.mean [inputs] [&amp;keys])</code>	$y = (1/N) \sum_{i=0}^{N-1} x_i$	average of input values
<code>(.min [inputs] [&amp;keys])</code>	$y = \min_{i=0}^{N-1} x_i$	min of input values
<code>(.mul [inputs] [&amp;keys])</code>	$y = \prod_{i=0}^{N-1} x_i$	multiply input values
<code>(.pwr [inputs] [&amp;keys])</code>	$y = x_0^{x_1^{x_2 \dots}}$	series of powers
<code>(.sub [inputs] [&amp;keys])</code>	$y = x_0 - \sum_{i=1}^{N-1} x_i$	subtract input values

#### Inputs:

The number of inputs or the input objects can be controlled by the parameters immediately following the block name function or by the keyword `:inputs` (default 2 inputs). If the keyword `:inputs` is used, the inputs created are unconnected and need to be connected afterwards by the chaining function `->`, for example as:

```
(defpatch p ((add (.add))) ;;; equal to (.add :inputs 2)
  (-> (.var 1.0) add (.probe "out"))
  (-> (.var 2.0) (in add 1)))
```

A more direct way is to specify explicitly the input objects as parameters to the block-making function, for example by making an adder patch:

```
(defpatch p ((x (.var 2.0))
  (add (.add 1.0 x t)))
  (-> (.var 3.0) (in add 2) (.probe "out")))
```

where the first input in the form `(.add 1.0 x t)` is a constant 1.0, the second one will be connected to the output of block `x`, and the third input is created but left unconnected until the chaining function form. The alternatives for input specification are (1) constant numeric value, (2) a block, in which case the first output of it is used, (3) specific output of a block, e.g. by form `(out block index)`, or (4) Lisp symbol `t` to make an unconnected input.

Inputs to these blocks may be scalars, vectors, or matrices. The dimensionality of inputs must match, i.e., being the same for all inputs. An exception from this rule is when one or

more of the inputs are scalars, whereby they are automatically expanded to the dimensionality of non-scalar inputs, which must match. An example thereof is

```
(defpatch p ((in1 (.var '(1.0 2.0 3.0 4.0)))
             (in2 (.var '(1.0 2.0 3.0 4.0)))
             (in3 (.var 2.0)))
  (-> (.add in1 in2 in3)))
```

### A.4.2 Single input blocks

Table A.2 lists the mathematical functions and operations with a single input. For example a block to negate the value of input is created simply by the form `(.neg)`, in which case the input is left open to be connected later in a patch definition. Another way is to give input  $x$  by `(.neg x)`.

Table A.2: Single-input math function blocks:  $y$  = output,  $x$  = input.

Block name and syntax	Operation	Description
<code>(.abs [in] [&amp;keys])</code>	$y =  x $	absolute value
<code>(.db [in] [&amp;keys])</code>	$y = 20 * \log_{10}( x )$	dB value (note 1)
<code>(.ceil [in] [&amp;keys])</code>	$y = \text{ceiling}(x)$	ceiling of $x$ ( <b>not yet</b> )
<code>(.cos [in] [&amp;keys])</code>	$y = \cos(x)$	cosine function
<code>(.cosh [in] [&amp;keys])</code>	$y = \cosh(x)$	hyperbolic cosine
<code>(.cot [in] [&amp;keys])</code>	$y = \cot(x)$	cotangent function
<code>(.exp [in] [&amp;keys])</code>	$y = \exp(x)$	exponent function
<code>(.floor [in] [&amp;keys])</code>	$y = \text{floor}(x)$	floor of $x$ ( <b>not yet</b> )
<code>(.inv [in] [&amp;keys])</code>	$y = 1/x$	reciprocal value
<code>(.log* [in] [&amp;keys])</code>	$y = (x \geq 0) ? \log(x) : 0$	log function (note 2)
<code>(.log10* [in] [&amp;keys])</code>	$y = (x \geq 0) ? \log_{10}(x) : 0$	log10 function (note 2)
<code>(.neg [in] [&amp;keys])</code>	$y = -x$	negation of input
<code>(.poly [in] [:c] [&amp;keys])</code>	$y = \sum c_i x^i$	polynomial ( <b>not yet</b> )
<code>(.rect [in] [&amp;keys])</code>	$y = (x \geq 0) ? x : 0$	half-wave rectify
<code>(.round [in] [&amp;keys])</code>	$y = \text{round}(x)$	rounding of $x$ ( <b>not yet</b> )
<code>(.sign [in] [&amp;keys])</code>	$y = \text{sign}(x)$	+1 for pos, else -1
<code>(.sin [in] [&amp;keys])</code>	$y = \sin(x)$	sine function
<code>(.sqr [in] [&amp;keys])</code>	$y = x^2$	square of input
<code>(.sqrt* [in] [&amp;keys])</code>	$y = (x \geq 0) ? \sqrt{x} : 0$	square root (note 3)
<code>(.tan [in] [&amp;keys])</code>	$y = \tan(x)$	tangent function
<code>(.tanh [in] [&amp;keys])</code>	$y = \tanh(x)$	hyperbolic tangent

#### Notes:

- 1) Abs value in decibels. Limited so that if  $\text{abs}(x) < 1.0e^{-30}$  then  $y = -600$  dB.
- 2) Negative inputs limited to zero. Avoid taking  $\log(0)$  which returns an error.
- 3) For negative  $x$  output  $y = 0$ .

### A.4.3 Other math blocks

A set of one- and two-input mathematical functions is presented in the following table.

Table A.3: Other math blocks.

Block name and syntax	Description
<code>(.coeff c [&amp;keys])</code>	Multiply by constant <i>c</i> (not function)
<code>(.mmax [input] [&amp;keys])</code>	maximum of input elements ( <b>not yet</b> )
<code>(.mmin [input] [&amp;keys])</code>	minimum of input elements ( <b>not yet</b> )
<code>(.mmul [in1] [in2] [&amp;keys])</code>	matrix multiply ( <b>not yet</b> )
<code>(.prod [input] [&amp;keys])</code>	product of elements in input ( <b>not yet</b> )
<code>(.sum [input] [&amp;keys])</code>	sum of elements in input
<code>(.transp [input] [&amp;keys])</code>	transpose a matrix/vector ( <b>not yet</b> )

### A.4.4 Predicate blocks for data comparison

The following set of functions take a variable number of inputs, no param inputs, and yield a logic valued output (zero = false, non-zero = true) corresponding to the result of the testing operation. Otherwise they are used like the multi-input arithmetic functions.

Trig/flag behavior ???

Table A.4: Multi-input predicate blocks.

Block name and syntax	Description
<code>(.eq [inputs] [&amp;keys])</code>	test: equal (=)
<code>(.ge [inputs] [&amp;keys])</code>	test: greater or equal ( $\geq$ )
<code>(.gt [inputs] [&amp;keys])</code>	test: greater than ( $>$ )
<code>(.le [inputs] [&amp;keys])</code>	test: less or equal ( $\leq$ )
<code>(.lt [inputs] [&amp;keys])</code>	test: less than ( $<$ )
<code>(.neq [inputs] [&amp;keys])</code>	test: not equal ( $\neq$ )

### A.4.5 Logic blocks

The following blocks are available for logic operations. All others than `.not` are multi-input blocks, with no param inputs, and one output. Inputs may have any numeric values: zero is interpreted as false and non-zero as true.

Trig/flag behavior ???

## A.5 Symbolic operations

Block compiler supports kind of symbolic algebra for manipulating expressions for transfer functions as well as for impedances and admittances. This means operations for computing with polynomials and rational forms as z-transform expressions. It supports also matrix operations with symbolic expressions. In addition to transfer function manipulation, symbolic operations

Table A.5: Logic operation blocks.

Block name and syntax	Description
(.and [inputs] [&keys])	operation: AND of inputs
(.or [inputs] [&keys])	operation: OR of inputs
(.nand [inputs] [&keys])	operation: NAND of inputs
(.nor [inputs] [&keys])	operation: NOR of inputs
(.not [input] [&keys])	operation: NOT of input
(.xor [inputs] [&keys])	operation: XOR (exclusive or)

are particularly useful in circuit synthesis of consolidated wave port elements and multiports. By convention, z-expression block names and operations start with underscore character, i.e., look like `_zxx`.

Table A.6: Symbolic expression operations.

Block name and syntax	Description
(.add [forms])	add expressions
(.sub form1 form2)	subtract expressions
(.mul [forms])	multiply expressions
(.div form1 form2)	divide expressions
(.neg form)	negate expression
(.inv form)	reciprocal of expression
(.zpoly [terms])	making of a z-polynomial
(.simpz zform)	simplify z-expression

### Operation-specific features:

#### • Polynomial make-function: `_zpoly`

A z-polynomial is created by form `(_zpoly c0 c1 c2 ...)`, where terms  $c_i$  will be the elements of the z-polynomial  $c_0z + c_1z^{-1} + c_2z^{-2} + \dots$ , or shortly  $\sum_{i=0}^{N-1} c_i z^{-i}$ . Terms  $c_i$  can be any numbers (including vectors) or blocks with numeric output (or such block outputs). For example `(_zpoly 1.0 (.var 1.0 'A))` makes a polynomial of two terms, `(.const 1.0)` and `(.var 1.0 'A)`, which can be characterized by  $1.0 + Az^{-1}$ .

#### • Addition of expressions: `_add`

??? Z-expressions can be added by form like `(_zadd H1 H2 H3 ...)`, where the number of parameters can be two or more. Mathematically addition of two forms can be characterized by  $H(z) = H_1(z) + H_2(z)$ . Addition of polynomials results in a polynomial, while the result will be a rational if any of the parameters is a rational.

#### • Multiplication of expressions: `_mul`

??? Z-expressions can be added by form like `(_zmul H1 H2 H3 ...)`, where the number of parameters can be two or more. Mathematically addition of two forms can be characterized by  $H(z) = H_1(z) \cdot H_2(z)$ . Multiplication of polynomials results in a polynomial, while



the result will be a rational if any of the parameters is a rational.

- **Subtraction of expressions: `_sub`**

??? Two Z-expressions can be subtracted by form like (`_zsub  $H_1$   $H_2$` ). Mathematically subtraction of two forms can be characterized by  $H(z) = H_1(z) - H_2(z)$ . Subtraction of polynomials results in a polynomial, while the result will be a rational if any of the parameters is a rational.

- **Division of expressions: `_div`**

??? Two Z-expressions can be divided by form like (`_zdiv  $H_1$   $H_2$` ). Mathematically division of two forms can be characterized by  $H(z) = H_1(z)/H_2(z)$ . Division of polynomials results in a rational, and the result will in general be a rational if any of the parameters is a rational.

- **Negation of expression: `_neg`**

??? A Z-expression can be negated by (`_zneg  $H_1$` ). Mathematically negation of a form can be characterized by  $H(z) = -H_1(z)$ . A polynomial results in a polynomial and a rational results in a rational.

- **Reciprocal of expression: `_inv`**

??? A Z-expression can be inverted to its reciprocal by (`_zinv  $H_1$` ). Mathematically, inversion of a form can be characterized by  $H(z) = 1/H_1(z)$ . In most cases the result is a rational.

- **Simplification of Z-expression: `_simpz`**

??? Z-expression are in most cases automatically simplified into minimal form. It can be done explicitly by (`_zinv  $H_1$` ). Simplification of a rational results in a polynomial if the denominator is constant 1.

## A.6 DSP blocks

Although BlockCompiler is developed particularly for physics-based modeling and simulation, a rich set of digital signal processing elements are need in practice. Therefore it includes a growing set of blocks that can be characterized as DSP blocks, such as signal generators, filters, delays, etc. This section presents a reference to these blocks and their use in modeling.

### A.6.1 Signal I/O blocks and data items

This set of blocks is related to inputting and outoutting signals. Some of these are to connect patches to external audio world and some are for controlling signal/param values during execution.

### A.6.2 Signal generation blocks

Table A.7: Signal I/O blocks.

Block name and syntax	Description
<code>(.ad [:channels] [&amp;keys])</code>	AD converter block ???
<code>(.const [:channels] [&amp;keys])</code>	constant block ???
<code>(.da [:channels] [&amp;keys])</code>	DA converter block ???
<code>(.trig [:channels] [&amp;keys])</code>	trigger block ???
<code>(.var [value] [name] [&amp;keys])</code>	variable block ???

Table A.8: Signal generation blocks.

Block name and syntax	Description
<code>(.imp [:gain] [:trig] [&amp;keys])</code>	triggerable impulse ???
<code>(.noise [:freq] [:gain] [&amp;keys])</code>	white noise generator ???
<code>(.pink [:freq] [:gain] [&amp;keys])</code>	pink noise generator ???
<code>(.ramp ??? [&amp;keys])</code>	linear ramp generator ???
<code>(.rtable ??? [&amp;keys])</code>	readtable (wavetable) ???
<code>(.sawtooth-osc [:freq] [:gain] [&amp;keys])</code>	sawtooth oscillator ???
<code>(.sin-osc [:freq] [:gain] [&amp;keys])</code>	sine wave oscillator ???
<code>(.sweep-lin [:f1] [f:2] [:time] [&amp;keys])</code>	linear sine sweep oscillator ???
<code>(.sweep-log [:f1] [f:2] [:time] [&amp;keys])</code>	log sine sweep oscillator ???

### A.6.3 Signal routing

These blocks are intended to control the signal flow: to convert between different scalar, vector, and matrix representations statically or to do some run-time routing operations.

#### Class-specific features:

- **Conditional execution** (`.cond`)

xxx

- **Gating of data flow** (`.gate`)

xxx

### A.6.4 Digital filter blocks

The following blocks are single-input single-output blocks for digital filtering. The specific keywords denoted by `[:xxx]` are for parameter controls (see below), and `[&keys]` are generic keywords common for all of them. The following table lists the available (and planned) filter blocks, which is followed by more specific description of each block class.

#### Specific keywords:

are class-dependent for specifying the param inputs of the blocks. If a keyword value is not given, the corresponding param input (with corresponding name) is created. For example a three-tap FIR filter with unit impulse input and probe at output can be made by:

Table A.9: Signal routing blocks.

Block name and syntax	Description
<code>(.cond [in] [:cond] [&amp;keys])</code>	execute block conditionally ???
<code>(.gate [inputs] [:gate] [&amp;keys])</code>	gating & sync of signals
<code>(.get [in] [:ind] [&amp;keys])</code>	get from register (sequentially) ???
<code>(.latch [inputs] [&amp;keys])</code>	triggered gating & sync of signals ???
<code>(.mix [inputs] [:gains] [&amp;keys])</code>	mix inputs by gains ???
<code>(.mix2 [in1] [in2] [:pan] [&amp;keys])</code>	mix two channels by pan factor ???
<code>(.pack [inputs] [&amp;keys])</code>	make vector from scalars
<code>(.packm [inputs] [&amp;keys])</code>	make matrix from vectors ???
<code>(.pan [in] [:pan] [&amp;keys])</code>	pan a sig to two outputs ???
<code>(.pick [in] [:ind] [&amp;keys])</code>	pick from vector to scalar/vector ???
<code>(.pickm [in] [:ind] [&amp;keys])</code>	pick from matrix to matrix/vector ???
<code>(.put [in] [:ind] [&amp;keys])</code>	put register input (sequentially) ???
<code>(.select [in] [:ind] [&amp;keys])</code>	select 1 from a vector dynamically ???
<code>(.select-1 [in] [:ind] [&amp;keys])</code>	select 1 from a vector statically ???
<code>(.shift-in [in] [:ind] [&amp;keys])</code>	shift register input (to vector) ???
<code>(.shift-out [in] [:ind] [&amp;keys])</code>	shift register output (from vector) ???
<code>(.unpack [in] [&amp;keys])</code>	make scalars from vector ???

```
(defpatch fp ((fir (.fir))) ;;; FIR block
  (-> (.var '(1.0 2.0 1.0)) (param fir)) ;;; coeffs -> param
  (-> (.impl) fir (.probe "out"))) ;;; impulse -> fir -> out
```

The same can be created simply by

```
(defpatch firpatch ()
  (-> (.fir (.impl) :coeffs '(1.0 2.0 1.0)) (.probe "out")))
```

where the input (optional) and the filter coefficients (keyword) are given directly to the FIR filter block.

### Class-specific features of digital filters:

#### • Allpass filter, first order (`.ap1`)

Filter block `.ap1` implements an allpass filter of first order with transfer function  $H(z) = (a - z^{-1}) / (1 - az^{-1})$ . Here  $a$  is the coefficient to control the group delay of the filter, while the magnitude response is flat (0 dB) at all frequencies.

Keyword `:a` is used to control the delay parameter: if not given, a param input with name `a` is created; if given, the value (number, list, or block object) is used to control `a`.

Input, output, and `:a` can be scalars, vectors, or matrices of equal dimensionality.

#### • Allpass cascade (`.ap1`)

NOT IMPLEMENTED YET

Table A.10: Digital filter blocks.

Block name and syntax	Description
<code>(.ap1 [in] [:a] [&amp;keys])</code>	1st order allpass filter
<code>(.ap1-ser [in] [:a] [&amp;keys])</code>	1st order allpass cascade ???
<code>(.bq [in] [:bs] [:as] [&amp;keys])</code>	biquad filter ???
<code>(.bq-par [in] [:bs] [:as] [&amp;keys])</code>	biquads in parallel ???
<code>(.bq-ser [in] [:bs] [:as] [&amp;keys])</code>	biquad cascade ???
<code>(.bp2 [in] [:coeffs] [&amp;keys])</code>	2nd order bandpass filter ???
<code>(.bp2-par [in] [:coeffs] [&amp;keys])</code>	.bp2 filters in parallel ???
<code>(.diff [in] [:coeff] [&amp;keys])</code>	differentiator
<code>(.diffb [in] [:coeff] [&amp;keys])</code>	bilinear differentiator
<code>(.fir [in] [:coeffs] [&amp;keys])</code>	FIR filter
<code>(.iir [in] [:bs] [:as] [&amp;keys])</code>	IIR filter (Direct Form II)
<code>(.integ [in] [:value] [:delayed] [&amp;keys])</code>	integrator
<code>(.integb [in] [:value] [&amp;keys])</code>	bilinear integrator
<code>(.lp1 [in] [:f] [:freq] [:gain] [&amp;keys])</code>	1st order lowpass filter
<code>(.lp2 [in] [:f] [:freq] [:q] [&amp;keys])</code>	2nd order lowpass filter ???
<code>(.sfir [in] :size [:coeffs] [:pos] [&amp;keys])</code>	sparse FIR filter
<code>(.wfir [in] [:coeffs] [&amp;keys])</code>	warped FIR filter ????
<code>(.wiir [in] [:bs] [:as] [&amp;keys])</code>	warped IIR filter ????

- **Biquad filter (.bq)**

Block class `.biquad` implements a general second order IIR filter. NOT DOCUMENTED YET

- **Biquad parallel filterbank (.bq-par)**

NOT DOCUMENTED YET

- **Biquad cascade (.bq-ser)**

NOT IMPLEMENTED YET

- **Bandpass filter, second order (.bp2)**

Filter block `.bp2` implements a second-order bandpass IIR filter. NOT IMPLEMENTED YET

- **Differentiator (.diff)**

Block `.diff` implements (impulse-invariant) differentiation of transfer function  $H(z) = f_s(1 - z^{-1})$ , where  $f_s$  is sample rate. The block approximates continuous-time derivative  $y(t) = dx(t)/dt$ , where  $y(t)$  is output and  $x(t)$  is input.

Keyword `:coeff` can be used for 'partial' differentiation, i.e.,  $H(z) = f_s(1 - cz^{-1})$ , where  $c$  is the controlled coefficient (if not given, default is 1.0).

Input and output (and `:coeff`) can be scalars, vectors, or matrices of equal dimensionality.

#### • Differentiator, bilinear (`.diffb`)

Block `.diff*` implements a bilinearly mapped differentiator with transfer function  $H(z) = 2f_s(1 - z^{-1})/(1 + z^{-1})$ . It has a pole at the Nyquist frequency  $z = -1$ , which may be problematic in some cases. Otherwise `.diff*` corresponds to block `.diff`.

#### • FIR filter (`.fir`)

Filter block `.fir` implements non-recursive FIR (finite impulse response) filtering (see Section ??) by computing the running convolution sum (output  $y(n)$ ) of the input ( $x(n)$ ) and the vector of filter coefficients representing the impulse response ( $h(i)$ ) of the filter. The order of the filter is  $N-1$ . Both input and output must be scalar-valued.

Filter coefficients can be given using the keyword `:coeffs` as a list, e.g., `(.fir :coeffs '(1.0 2.0 1.0))`, or as a vector (output) of another block, e.g., `(.fir :coeffs (.var '(1.0 2.0 1.0)))`. If the keyword `:coeffs` is not used, a param input (named `coeffs`) is created, which has to be connected to a block specifying the filter coefficients. The coefficients can be changed at any time (assuming they are not given as a constant vector), so the filter can be time-varying. The input of the FIR filter can be given as the first (optional) parameter, such as `(.fir (.impl))`. If not given, as in the examples before, an input terminal for connection is created.

#### • IIR filter (`.iir`)

Block class `.iir` implements recursive (infinite impulse response) filtering. It is implemented as a Direct Form II filter (see Section ??).

The following keywords are available for filter coefficient control: `:bs` for vector of numerator coefficients (length  $\geq 1$ ) and `:as` for vector of denominator coefficients (length  $\geq 2$ ).

IIR filters are used like the FIR filter, except the different coefficient keywords. The first denominator coefficient `as[0]` doesn't have to be 1.0.

#### • Integrator (`.integ`)

Block `.integ` implements (impulse-invariant) discrete-time integration that approximates continuous-time integration of  $y(t) = y(0) + \int_0^t x(\tau) d\tau$ . For example for constant input value of 1.0 the integrator output ramps from 0.0 to value 1.0 in 1.0 second.

Keywords available are `:value` for setting the initial output,  $y(0)$ , and `:delayed` for values `t` or `nil` for controlling if the output contains a delay or not. For the non-delayed case (default) the transfer function is  $H(z) = (1/f_s)/(1 - z^{-1})$  and for the delayed case it is  $H(z) = (1/f_s)z^{-1}/(1 - z^{-1})$ , where  $f_s$  is the sample rate of computation.

Input and output can be scalars, vectors, or matrices of equal dimensionality.

#### • Integrator, bilinear (`.integb`)

Block `.integ*` implements bilinearly mapped discrete-time integration with transfer func-

tion  $H(z) = (1/2f_s)(1 + z^{-1})/(1 - z^{-1})$ . The difference to (non-delayed) `.integ` is the zero at the Nyquist frequency  $z = -1$ . Block `.integ*` is used like `.integ` except that keyword `:delayed` is not available (no delay element between input and output).

#### • Lowpass filter, first order (`.lp1`)

Filter block `.lp1` implements the first order lowpass IIR filter with transfer function of  $H(z) = g(1 - a)/(1 - az^{-1})$ , where  $g$  is the DC gain, controlled by keyword `:gain`, and  $a$  is the denominator coefficient of  $z^{-1}$ , controlled by keyword `:a`. The response is flat for  $a = 0.0$  and turns increasingly to lowpass filter for values towards  $a \rightarrow 1.0$ . The filter works also as slight high-frequency boosting for negative values of  $a$ . The filter is stable for  $-1.0 < a < 1.0$ . If `:a` (and `:freq`, see below) is not given, a param input with name `a` is created. The same happens if the keyword `:a` is given value `t`.

Another choice of controlling the frequency response is to use keyword `:freq` instead of `:a`. Then the lowpass cutoff frequency will be  $f$ , where  $f$  is the value given to the keyword. This approximation of cutoff frequency is accurate only for  $f \ll f_s/2$ , where  $f_s$  is the sample rate. (For high values of  $f$  the filter may become unstable.) A param input with name `freq` is created when the keyword `:freq` is given value `t` and keyword `:a` is not specified.

Param `:gain` is a linear DC gain factor. If it is `nil` or not given, the DC gain of the filter is 1.0. If it is given symbol value `t`, a param input with name `gain` is created.

Input, output, and params can be scalars or vectors of equal dimensionality.

#### • Lowpass filter, second order (`.lp2`)

Filter block `.lp2` implements a second-order lowpass IIR filter. NOT IMPLEMENTED

#### • Sparse FIR filter (`.sfir`)

Block `.sfir` implements a sparse-tap FIR filter. In addition to filter coefficients it requires same size of index vector to specify the positions of the non-zero tap coefficients.

Keyword `:coeffs` is used to give the coefficients as in FIR filter; if it is not specified, a param input with name `coeffs` is created. Keyword `:pos` is used to give the tap positions; if not given, a param input with name `pos` is created. Keyword `:size` is required to specify the total length of the filter. Tap positions must be within this range.

Input and output have to be scalar (one-dimensional); `coeffs` and `pos` must be vectors of same size.

### A.6.5 Delay blocks

Delays are an inherent part of physics-based modeling and DSP. In physics-based modeling, fractional delays (see Section ??), often time varying with delay control, are needed. The following set of delays is available as DSP blocks for BlockCompiler. Delays are single-input single-output blocks. Unit delays `.d` and `.d-` can have vector- and matrix-valued inputs and outputs, other delays are single-channel (scalar-valued) delays.

#### Specific features:

xxxx

Table A.11: Delay blocks.

Block name and syntax	Description
<code>(.d [:value] [&amp;keys])</code>	unit delay
<code>(.d- [:value] [&amp;keys])</code>	unit delay with sign negation
<code>(.delay [*keys] [&amp;keys])</code>	controllable delay ???
<code>(.dn [length] [&amp;keys])</code>	integer length delay ???
<code>(.tap-delay taplist [&amp;keys])</code>	multitap controllable delay ???

### A.6.6 Miscellaneous DSP blocks

xxx

## A.7 WDF elements

Wave digital elements have been described in Chapter 4. They form a large part of physical modeling blocks in BlockCompiler, particularly when lumped element models are to be created. The WDF blocks and constructs available in BlockCompiler are listed in this reference section.

### A.7.1 Lumped linear WDF one-port elements with I-port

The following table lists the basic one-port lumped elements used in WDF models. There are different versions for each physical domain, although the electric ones (.R, .C, .L, etc. can be used for other domains by keyword :domain, e.g., (.R 1.0 :domain 'mechanic.

Value can be numeric value, explicit constant such as (.R (.const 1.0)), a variable such as (.R (.var 1.0)), or any block with proper output.

Table A.12: Lumped linear one-port WDF elements with I-port.

Block name and syntax	Description
(.R [val] [&keys])	electrical resistance
(.Rm [val] [&keys])	damper, mechanical resistance
(.Ra [val] [&keys])	acoustical resistance
(.G [val] [&keys])	electrical conductance
(.Gm [val] [&keys])	mechanical conductance
(.Ga [val] [&keys])	acoustical conductance
(.C [val] [&keys])	electrical capacitance
(.Cm [val] [&keys])	compliance, mechanical capacitance
(.Ca [val] [&keys])	acoustical capacitance
(.L [val] [&keys])	electrical inductance
(.Lm [val] [&keys])	mass, mechanical inductance
(.Lm [val] [&keys])	acoustical inductance
(.E [val] [rs] [&keys])	electrical voltage source
(.Fm [val] [rs] [&keys])	mechanical force source
(.Pa [val] [rs] [&keys])	acoustical pressure source
(.J [val] [rs] [&keys])	electrical current source
(.Vm [val] [rs] [&keys])	mechanical velocity source
(.Va [val] [rs] [&keys])	acoustical volume velocity source



### A.7.2 Lumped linear WDF one-port elements with T- port

These WDF elements are can be used only as rool elements of WDF trees:

Table A.13: Lumped linear WDF one-port elements with T- port.

Electrical	Mechanical	Acoustical	Description
<code>.R*</code>	<code>.Rm*</code>	<code>.Ra*</code>	resistance
<code>.C*</code>	<code>.Cm*</code>	<code>.Ca*</code>	capacitance
<code>.L*</code>	<code>.Lm*</code>	<code>.Lm*</code>	inductance

### A.7.3 RLC-combination one-ports

The following one-port elements are consolidated implementations of basic combinations of resistors, capacitors, and inductors. They are not optimized by efficiency; depending on case, they might be faster or slower than circuits composed by `.ser` and `.par` adaptor constructions. Parameters may be given by keywords `:R`, `:L`, and `:C`, or left unspecified, whereby corresponding param inputs will be created for the block.

- `(.RCser [ :R ] [ :C ])`, resistor and capacitor in series.
- `(.RCpar [ :R ] [ :C ])` resistor and capacitor in parallel.
- `(.RLser [ :R ] [ :L ])` resistor and inductor in series.
- `(.RLpar [ :R ] [ :L ])` resistor and inductor in parallel.
- `(.LCser [ :L ] [ :C ])` inductor and capacitor in series.
- `(.LCpar [ :L ] [ :C ])` inductor and capacitor in parallel.
- `(.LCRser [ :L ] [ :C ] [ :R ])` Not yet there
- `(.LCRpar [ :L ] [ :C ] [ :R ])` Not yet there

### A.7.4 Symbolic z-expression elements

In symbolic manipulation of z-expressions to make arbitrary one-port consolidated elements the following basic element can be utilised. Value parameter can be numeric value, a `.const` object, a `.var` object, or any block with proper output value. These are NOT physical port elements but forms that can be symbolically manipulated or converted to wave-port elements by `.Z`, see the next subsection.

Table A.14: Z-expression generation for basic circuit elements for symbolic manipulation.

Block name and syntax	Description
<code>(_R value)</code>	bilinearly mapped resistance
<code>(_C value)</code>	bilinearly mapped capacitance
<code>(_L value)</code>	bilinearly mapped inductance

### A.7.5 Consolidated one-port blocks

This is an important set of elements for creating complex WDF models. They are particularly useful when an impedance or admittance is given in z-transform expression, not as an analog circuit model. These blocks are NOT physical

Table A.15: Consolidated one-port blocks.

Block name and syntax	Description
(.Z z-expression)	one-port from z-expression
(.Z2 [:a1] [:a2] [:b0] [:b1] [:b2])	2nd order impedance block
(.Y2 [:a1] [:a2] [:b0] [:b1] [:b2])	2nd order admittance block
(.Z2ser [:a1] [:a2] [:b0] [:b1] [:b2])	series 2nd order impedances
(.Y2par [:a1] [:a2] [:b0] [:b1] [:b2])	parallel 2nd order admittances
(.Zpoly [coeffs])	impedance in polynomial form
(.Ypoly [coeffs])	admittance in polynomial form
(.Zratio [coeffs])	impedance in rational form
(.Yratio [:bs] [:as])	admittance in rational form
(.ZFQ-modes [:Z] [:F] [:Q])	Modal impedances in series
(.YFQ-modes [:Y] [:F] [:Q])	Modal admittances in parallel

#### Using consolidated one-ports:

- .Z, making a one-port consolidated element from a z-expression (see A.5, page 212, and ??, page ??). This is a powerful feature since the impedance and its parametric control can be derived symbolically and then converted to one-port element. However, computational cost, especially parameter updating, becomes easily expensive.
- .Z2, second-order rational expression for wave-port impedance:  
(.Z2 [:a1] [:a2] [:b0] [:b1] [:b2]), where the keyword parameters correspond to z-transform expression

$$Z(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

- .Y2, second-order rational expression for wave-port admittance:  
(.Y2 [:a1] [:a2] [:b0] [:b1] [:b2]), where the keyword parameters are used as for block .Z2.
- .Z2ser, second-order rational expression for impedances in series:  
(.Z2ser [:a1] [:a2] [:b0] [:b1] [:b2]). Each keyword is given as a vector (of the same dimension  $N$ ), where each element corresponds to a parameter in one of the series-connected impedances:

$$Z(z) = \sum_{i=0}^{N-1} \frac{b_{0,i} + b_{1,i} z^{-1} + b_{2,i} z^{-2}}{1 + a_{1,i} z^{-1} + a_{2,i} z^{-2}}$$

- .Y2par, second-order rational expression for admittances in parallel:  
(.Y2par [:a1] [:a2] [:b0] [:b1] [:b2]). Keywords are used as for block .Z2ser.

- `.Zpoly`, impedance given in polynomial form:  
(`.Zpoly [coeffs]`) or (`.Zpoly [:coeffs]`), by vector `coeffs` consisting of coefficients  $b_i$ , corresponding to

$$Z(z) = \sum_{i=0}^{N-1} b_i z^{-i}$$

- `.Ypoly`, admittance given in polynomial form:  
(`.Ypoly [coeffs]`) or (`.Ypoly [:coeffs]`), in a way similar to `.Zpoly`.
- `.Zratio`, impedance given in rational expression:  
(`.Zratio [:bs] [:as]`), with numerator ( $b_i$ , order  $N$ ) and denominator ( $a_i$ , order  $M$ ) coefficients, corresponding to

$$Z(z) = \frac{\sum_{i=0}^{N-1} b_i z^{-i}}{\sum_{i=0}^{M-1} a_i z^{-i}}$$

- `.Yratio`, admittance given in rational expression:  
(`.Yratio [:bs] [:as]`), in a way similar to `.Zratio`.
- `.ZFQ-modes, ???` (`.ZFQ-modes [:Z] [:F] [:Q]`) (modal impedance synthesis)
- `.YFQ-modes, ???` (`.YFQ-modes [:Y] [:F] [:Q]`) (modal admittance synthesis)

### A.7.6 Two-port elements

Two-port elements are used to connect together subcircuits for various transformation between them. All two-port elements in the following list are such where the first port is of type I-port and second port is a T-port, i.e., a reflection-free port.

- (`.xformer N`) creates an ideal transformer with turns-ratio  $N$ , see Section ??.
- (`.xform item N`) xxxx
- (`.gyrator ?`) xxxx
- (`.gyrate item`) xxx
- `.dualizer` is a two-port that converts between dual-models, i.e., impedance and admittance ports. Form (`.dualizer`) creates a dualizer object where port 0 is an I-port and port 1 is a T-port, so that port resistance is propagated in the direction from port 0 to port 1.
- (`.dual item`) xxx
- `.xducer / .xduce` ????

### A.7.7 Probe elements

-

`.across`  
`.through`

.voltage  
 .current  
 .force  
 .velocity  
 .displacement  
 .acceleration  
 .pressure  
 .flow  
 .power  
 .energy ?  
 GENERAL:  
 (port block index)  
 connect ???  
 NONLINEAR:  
 .diode

## A.8 Connectivity for circuits and networks

-

.ser  
 .par  
 .par2  
 .root  
 .ser-adapt (?)  
 .ser3-adapt  
 .par-adapt (?)  
 .par3-adapt  
 .par2-adapt

## A.9 Circuit synthesis

x

## A.10 DWG elements

???

## A.11 FDTD elements

???

**A.11.1 Misc elements**

K/W-conversion

...

# Index

.C, 221  
.C\*, 222  
.Ca, 221  
.Cm, 221  
.E, 221  
.Fm, 221  
.G, 221  
.Ga, 221  
.Gm, 221  
.J, 221  
.L, 221  
.L\*, 222  
.La, 221  
.Lm, 221  
.Pa, 221  
.R, 221  
.R\*, 222  
.Ra, 221  
.Rm, 221  
.Va, 221  
.Vm, 221  
.Y2, 223  
.Y2par, 223  
.YFQ-modes, 223  
.Ypoly, 223  
.Yratio, 223  
.Z2, 223  
.Z2ser, 223  
.ZFQ-modes, 223  
.Zpoly, 223  
.Zratio, 223  
.abs, 211  
.ad, 215  
.ap1, 217  
.ap1-ser, 217  
.bp2, 217  
.bp2-par, 217  
.bq, 217  
.bq-par, 217  
.bq-ser, 217  
.ceil, 211  
.coeff, 212  
.const, 215  
.cos, 211  
.cosh, 211  
.cot, 211  
.d, 220  
.d-, 220  
.da, 215  
.db, 211  
.delay, 220  
.diff, 217  
.diffb, 217  
.dn, 220  
.exp, 211  
.fir, 217  
.floor, 211  
.gate, 216  
.get, 216  
.if, 216  
.iir, 217  
.imp, 215  
.integ, 217  
.integb, 217  
.inv, 211  
.latch, 216  
.log\*, 211  
.lp1, 217  
.lp2, 217  
.mix, 216  
.mix2, 216  
.mmax, 212  
.mmin, 212  
.mmul, 212  
.neg, 211  
.noise, 215  
.pack, 216  
.packm, 216

- .pan, 216
- .pick, 216
- .pickm, 216
- .pink, 215
- .poly, 211
- .probe, 163
- .prod, 212
- .put, 216
- .ramp, 215
- .rect, 211
- .round, 211
- .rtable, 215
- .sawtooth-osc, 215
- .select, 216
- .select-1, 216
- .sfir, 217
- .shift-in, 216
- .shift-out, 216
- .sign, 211
- .sin, 211
- .sin-osc, 215
- .sqr, 211
- .sqrt\*, 211
- .sum, 212
- .sweep-lin, 215
- .sweep-log, 215
- .tan, 211
- .tanh, 211
- .tap-delay, 220
- .transp, 212
- .trig, 215
- .unpack, 216
- .var, 215
- .wfir, 217
- .wiir, 217
- \_C, 222
- \_L, 222
- \_R, 222
- T-port, 96, 106
- 1-D, 13
- 2-D, 13
- 3-D, 13
- A/D-conversion, 66
- across variable, 12
- active, 15
- adapted port, 106
- adaptive systems, 71
- admittance, 12, 26
- aliasing, 14, 66, 131
- aliasing distortion, 131
- all-pole filter, 69
- allpass filter, 217
- Ampere, 25
- amplitude, 34, 55
- amplitude envelope, 35
- analog signal, 55
- analog signal processing, 14, 55
- analog-to-digital conversion, 66
- analogy, 12
- anechoic chamber, 44
- angular frequency, 34, 55
- antiformant, 50
- AR-modeling, 69
- articulation, 49
- autocorrelation, 64
- autocorrelation method, 69
- autoregressive modeling, 69
- BöppckCompiler, 159
- base band, 14
- bass-reflex loudspeaker, 43
- BC, 159
- binary tree, 112
- biquad filter, 218
- black box, 58
- Blackman window, 62
- block, 161
- block inputs, 205
- block outputs, 205
- block params, 205
- block ports, 205
- block-based, 16
- block-item, 204
- block-structure, 205
- boundary element method, 17
- breathy phonation, 49
- c-code, 205
- capacitance, 27, 141
- capacitor, 27
- cardioid microphone, 46
- causal, 59
- causality, 13
- cepstrum, 66

- chaotic, 16
- circuit, 30
- circuit analogy, 51
- circulator, 30, 115
- closed box loudspeaker, 43
- complex number, 59
- complex-valued function, 59
- complexity, 16
- condenser microphone, 46
- conductance, 26
- constriction, 50
- continuous-time signal, 55
- convolution, 58
- convolution integral, 59
- convolution sum, 59
- creaky voice, 49
- cross-correlation, 64
- current, 25
  
- D/A-conversion, 66
- dashpot, 34
- defpatch, 161
- delay, 220
- delay-free loop, 13
- deterministic, 16
- DFT, 15, 60
- differential capacitance, 142
- differential equations, 22
- differentiator, 218
- digital filtering, 68
- digital signal, 55
- digital signal processing, 14, 55, 66
- digital signal processor, 66
- digital-to-analog conversion, 66
- diode, 104
- discrete Fourier transform, 15, 60
- discrete-time modeling, 14
- discrete-time signal, 55
- displacement, 33
- distortion, 57
- distributed model, 13
- distributed nonlinearity, 130
- domain, 12
- DSP, 14, 55, 66
- dual variables, 12
- dualizer, 30, 115, 224
- dynamic microphone, 46
- dynamic system, 13
  
- eigenmode, 36
- electret microphone, 46
- electromagnetic transducer, 40
- electrostatic headphones, 44
- electrostatic transducer, 45
- electrodynamic transduction, 119
- energetic behavior, 15
- energy, 15
- energy conservation, 15
- envelope, 62
- equivalent model, 12
- Euler relation, 59
- excitation, 35
  
- fast Fourier transform, 60
- FFT, 15, 60
- finite element method, 17
- FIR filter, 68, 218
- flow variable, 12
- force, 33
- formant, 50
- Fourier analysis, 60
- Fourier synthesis, 60
- Fourier transform, 15, 60
- frequency, 34, 55
- frequency domain, 57, 59
- frequency warping, 100, 131
- fundamental frequency, 49
  
- glottis, 49
- group delay, 62
- gyrator, 29
  
- Hamming window, 62
- hanning window, 62
- hard nonlinearity, 130
- headphones, 44
- Helmholtz resonator, 39
- hop size, 63
  
- ideal transformer, 29, 224
- IIR filter, 68, 218
- image source method, 17
- immittance, 26
- impedance, 12, 26
- impulse response, 57
- incident wave, 13



- inductance, 28
- inductor, 28
- instantaneous phase, 35
- integral transforms, 23
- integrator, 219
- interconnection, 111
- intermodulation, 15
- inverse filter, 70
- K-variables, 13
- Kaiser window, 62
- Kelly-Lochbaum junction, 82
- keyword syntax, 163
- keywords, 163
- Kirchhoff, 13
- Kirchhoff current law, 30
- Kirchhoff laws, 30
- Kirchhoff voltage law, 31
- KL-junction, 82
- Laplace transform, 15, 60
- larynx, 49
- learning systems, 71
- linear, 57
- linear combination, 37
- linear prediction, 69
- linear predictive coding, 69
- linearity, 14
- Log-PCM, 67
- longitudinal, 36
- loop, 30
- loss, 35
- lossless, 130
- lowpass filter, 219
- LP analysis, 69
- LP spectrum, 70
- LTI, 15, 57
- lumped element, 26
- lumped model, 13
- magnitude spectrum, 61
- make-function, 161
- mass, 33
- Matlab, 159
- medium, 35
- microphone, 45
- microphone array, 46
- mobility, 12
- mode, 36
- modeling paradigm, 17
- modularity, 16
- mucosa, 49
- mutual inductance, 28
- nasal cavity, 49
- nasal tract, 49
- neutral vowel, 52
- node, 30
- noise, 38, 55
- non-LTI modeling, 129
- nonadapted capacitance, 222
- nonadapted inductance, 222
- nonadapted resistance, 222
- nonlinear, 57, 59
- nonlinear capacitance, 141
- nonlinear distortion, 57
- nonlinearity, 129
- norator, 30
- normal mode, 36
- nullator, 30
- Nyquist frequency, 14, 66
- Nyquist theorem, 66
- object-oriented, 16
- one-port element, 26
- oral cavity, 49
- overcomplexity, 16
- oversampling, 131, 132
- parallel connection, 31
- parallel junction, 79
- parameter input, 162
- parameter iteration, 131
- parameter propagation, 130
- parametric control, 130
- partial, 60
- passive, 130
- passive element, 26
- patch, 161, 203
- patch compilation, 205
- patch creation, 203
- patch loading, 205
- patch running, 206
- patch stepping, 205
- patch structure, 204
- PCM coding, 67

- pd, 159
- perfect reconstruction, 14
- pharynx, 49
- phase delay, 62
- phase spectrum, 61
- phase unwrapping, 62
- phonation, 49
- physical domain, 12
- physics, 21
- port, 16, 25
- position, 33
- potential variable, 12
- power, 15, 26
- pressure microphone, 46
- probe, 163
- pure tone, 55
- PureData, 159
  
- quantization, 67
- quantization noise, 67
- QUARL, 116
  
- radiation impedance, 51
- ray tracing, 17
- reciprocity, 14
- rectangular window, 62
- reflectance, 96
- reflected wave, 13
- reflection coefficient, 82
- reflection-free port, 106
- residual signal, 70
- resistance, 26
- resistor, 26, 27
- resonance, 35
- resonator, 35
  
- sampling rate, 14
- sampling theorem, 14, 66
- scattering, 79
- series connection, 31
- short-time Fourier analysis, 63
- signal, 14, 55
- signal input, 162
- signal processing, 14, 55
- signal-to-noise ratio, 67
- sinusoidal oscillation, 34
- smooth nonlinearity, 130
- soft palate, 50
  
- sound, 55
- sound pressure, 51
- source, 35
- spatial sampling, 14
- spatial aliasing, 14
- spectral envelope, 70
- spectrogram, 63
- spectrum, 61
- spectrum analysis, 61
- spring constant, 34
- stability, 15
- stable, 15
- static system, 13
- stochastic, 16
- Sturm-Liouville transform, 15
- superposition, 14
- synthesis filter, 70
  
- terminal, 16, 25
- through variable, 12
- time domain, 57
- time invariant, 57
- time variable, 55
- time variant, 59
- time-frequency representation, 63
- time-invariant, 15
- time-variant models, 129
- trainable systems, 71
- transducer, 40, 118
- transfer function, 57
- transformer, 28
- transmission line, 51
- transversal, 36
- tree structure, 111
- turns ratio, 29
- two-port, 51
  
- unit delay, 67
- unit element, 116
- unvoiced speech, 50
- uvula, 50
  
- vocal cords, 49
- vocal folds, 49
- vocal fry, 49
- vocal tract, 49
- voice, 55
- voiced sound, 50

Volt, 25  
voltage, 25  
Volterra filters, 71  
volume velocity, 51  
  
W-variables, 13  
Watt, 26  
Wave digital filters, 93  
wave digital filters, 13  
wave variables, 13  
wavelet, 64  
wavelet analysis, 63  
WDF, 93  
WDF elements, 221  
Wigner distribution, 63  
window function, 62  
windowing, 62  
  
z-transform, 67