Coast Model

Documentation Manual

By

James G. Norris, Troy Frever, and Susannah Iltis

Columbia Basin Research

University of Washington

Seattle, WA 98195

Prepared For:

U.S. Department of Commerce

National Oceanic and Atmospheric Administration

National Marine Fisheries Service

Northwest Fisheries Science Center

2725 Montlake Blvd. E.

Seattle, WA 98112-2097

April 2001

Table of Contents

1 Introduction 11

1.1 Background 11

1.2 Code Framework Overview 11

1.3 Specific Code Features 12

1.3.1 Discrete Time Chronology 12

1.3.2 Cohort Based 12

1.3.3 Processes and Data Controlled by “Managers” 12

1.3.4 Data Access by Generic Array 13

1.3.5 Data Request Manager 13

1.3.6 Input/Output Using Existing Tools 13

1.3.7 Multi-Timestep Iteration Capability 13

1.4 Code Limitations 13

1.5 Pacific Salmon Commission Chinook Model 14

1.5.1 General Description of the PSC Chinook Model 14

1.5.2 - Brief History of the PSC Chinook Model 16

2 Coast Model Processes 19

2.1 Computation Flow 19

2.1.1 Overview 19

2.1.2 PSC Chinook Model Implementation 19

2.1.3 Future Processes 20

2.2 Cohort Ageing 21

2.2.1 Overview 21

2.2.2 PSC Chinook Model Implementation 21

2.2.3 Future Ageing Processes 22

2.3 Natural Mortality 22

2.3.1 Overview 22

2.3.2 PSC Chinook Model Implementation ... 22

2.3.3 Future Natural Mortality Processes ... 23

2.4 Fishing Mortality ... 24

2.4.1 Overview ... 24

2.4.2 Legal Catches ... 24

2.4.2.1 SingleCohort ... 24

2.4.2.2 Multiple Cohorts (Safe-Guarded Secant Algorithm) ... 25

2.4.3 Incidental Mortalities ... 26

2.4.3.1 Shakers ... 26

2.4.3.1.1 Compute the "*StockWts*" ... 27

2.4.3.1.2 Compute the "*TotalPNV*" and "*TotalPV*" ... 27

2.4.3.1.3 Compute the "*EncounterRate*" ... 27

2.4.3.1.4 Compute the "*FracNV*" ... 27

2.4.3.1.5 Compute "TotalShakers" ... 27

2.4.3.1.6 Distribute "*TotalShakers*" by stock and age using the *FracNVs* ... 28

2.4.3.2 Chinook Non-Retention Mortalities ... 28

2.4.3.2.1 Compute the ratios relating legal CNR mortalities to the legal catch and sublegal CNR mortalities to the shakers. ... 28

2.4.3.2.2 Compute legal and sublegal CNR mortalities without considering multiple encounters ... 31

2.4.3.2.3 Adjust CNR mortalities for multiple encounters ... 31

2.5 Maturation ... 35

2.5.1 Overview ... 35

2.5.2 PSC Chinook Model Implementation ... 35

2.5.3 Future Maturation Processes ... 36

2.6 Spawning ... 36

2.6.1 Overview ... 36

2.6.1.1 LinearProduction ... 38

2.6.1.2 RickerProduction ... 39

2.6.1.3 EnhancedRickerProduction ... 39
2.6.1.4 VariableTruncatedRickerProduction ... 40
2.6.2 PSC Chinook Model Implementation ... 41
2.6.3 Future Spawning Processes ... 41
2.7 Migration ... 42
2.7.1 Overview ... 42
2.7.2 PSC Chinook Model Application ... 42
3 Input Language ... 45
3.1 Introduction ... 45
3.2 Token Types ... 45
3.2.1 Simple Tokens ... 45
3.2.2 Command Block Tokens ... 45
3.2.3 Generic Array Tokens ... 46
3.2.4 Subtokens ... 46
3.2.5 Special Tokens ... 46
3.3 Generic Arrays ... 46
3.3.1 Generic Array Subtokens ... 48
3.3.2 Generic Array Dimension Specifiers ... 49
3.4 Top Level Tokens ... 50
3.5 Configuration Tokens ... 50
3.5.1 Fishery Configuration Tokens ... 50
3.6 Stock Configuration Tokens ... 51
3.7 Harvest Rate Tokens ... 51
3.8 Ceiling Tokens ... 52
3.8.1 CeilingScalars ... 52
3.8.2 MultiTimeStepCeilingScalars ... 53
3.8.3 CeilingData ... 53
3.9 CNRData Tokens ... 54

3.9.1 CNR Methods 54
3.10 Cohorts 56
3.11 FPData Tokens 57
3.12 FisherySchedule Tokens 57
3.13 MaturationData Tokens 58
3.14 NatMortRateData Tokens 59
3.15 PnvData Tokens 59
3.16 ProductionFunctions Tokens 60
3.16.1 Production Function Types 60
3.17 ShakerData Tokens 62
3.17.1 Shaker Methods 62
3.17.2 VulnerabilityTable Tokens 63
3.18 TransitionMatrix Tokens 63
3.18.1 Transition Matrix Data 63
4 Output Language 65
4.1 Overview 65
4.2 CohortID 65
4.3 Output Sentences Supported by Coast Model 66
4.3.1 CABN sentence for cohort abundance 66
4.3.2 NMRT sentence for natural mortality 66
4.3.3 FMRT sentence for fishing mortality 67
4.4 Proposed Output Sentences For Future Use 67
4.4.1 CMIG sentence for cohort migration 67
4.4.2 SABN sentence for stock abundance in a region 68
4.4.3 CLHD sentence for cohort life history data 68
4.5 Generating the Output Data File 68
5 Code Description 69
5.1 Introduction 69

5.2 Naming Conventions 69
5.3 Class Overview 69
5.3.1 Monostates and Managers 69
5.3.2 Globals 70
5.3.3 Other Important Classes 70
5.4 Process Overview 70
5.4.1 Simulation Processes 70
5.5 Class and Object Detail 71
5.5.1 Managers and Other Globals 71
5.5.2 Common Fundamental Objects 71
5.5.2.1 Cohort and CohortID 72
5.5.2.2 GenericArray and GenericArrayIndex 72
5.5.2.3 GenericArrayFactory 75
5.5.2.4 Example: Adding a New Cohort Characteristic 76
5.5.2.5 Iteration 77
5.5.2.5.1 State 77
5.5.2.5.1.1 SystemState 78
5.5.2.5.2 IterationControl 78
5.5.2.5.3 IterationManager 79
5.5.2.6 DataRequest 79
5.5.2.7 LogMsg 79
5.5.3 Special Purpose Classes and Objects 80
5.5.3.1 Harvest Classes 80
5.5.3.1.1 FisheryUnit 80
5.5.3.1.2 FisheryPolicy 81
5.5.3.1.3 PolicyControl 81
5.5.3.1.4 HvMort and HarvestProcess 81
5.5.3.1.5 Legal Catch Process Detail 82

5.5.3.1.5.1 Harvest Computation Flow (Legal Catch) 83
5.5.3.1.6 Incidental Mortalities 84
5.5.3.1.6.1 Shaker 84
5.5.3.1.6.2 CNR 84
5.5.3.2 Production Classes 84
5.5.3.2.1 Production and ProductionFunction 84
5.5.3.2.2 CohortGenerator 85
5.5.3.3 Data Output 85
5.5.3.4 Data Input Parser 85
5.5.3.4.1 HarvestParser – An In-depth Example 86
5.5.3.4.1.1 HarvestParser Declarations 86
5.5.3.4.1.2 HarvestParser Definitions 87
5.5.3.4.1.3 Using the HarvestParser 90
5.6 Process Detail 90
5.6.1 Initialization and Cleanup 90
5.6.2 The Engine 91
Appendix A: Glossary 92
Appendix B: Examples 95
B.1 Introduction 95
B.2 Prototypes With No Harvest 95
B.2.1 Proto 0 (One chinook stock over 100 years) 96
B.2.2 Proto 1 (Like PSC chinook model, but only one stock) 96
B.2.3 Proto 1a (Like PSC chinook model, but simulating one coho stock) 96
B.2.4 Proto 2 (Add "estuary" and more ocean regions; stock distribution by ocean region) 97
B.2.5 Proto 3 (Add more terminal regions; stock distribution by region) 97
B.2.6 Proto 4 (Increase timesteps to 13; spread natural mortality over timesteps; migration during most timesteps) 97
B.2.7 Proto 5 (Maturation occurs during timesteps 4 to 7 in all ocean regions) 98

B.2.8 Proto 6 (same as Proto 5, but add the coho stock) ... 98

B.2.9 Proto 7 (each stock different natural mortality, maturation, and migration process) ... 98

B.3 Prototypes With Harvest ... 99

B.3.1 Proto 8 (Same as Proto 7, but add an ocean fishery) ... 99

B.3.2 Proto 9 (Same as Proto 8, but add an inside fishery that has a multi-phase catch ceiling that spans three timesteps) ... 99

B.3.3 Proto 10 (Same as Proto 8, but add a third fishery; all fisheries have multi-phase catch ceilings spanning five timesteps) ... 100

Appendix C: Discussion Papers ... 101

C.1 Overview ... 101

C.2 Ageing Process ... 101

C.2.1 Spring Stock Algorithms ... 101

C.3 Mortality Processes ... 102

C.3.1 State Space Model Considerations ... 102

C.3.1.1 Background ... 103

C.3.1.2 Harvest Process ... 103

C.3.1.3 Fishing Process ... 104

C.3.1.4 Code Issues ... 104

C.3.1.5 Code Solution ... 104

C.3.1.6 Code/Algorithm Problems ... 104

C.3.1.7 SSM Parameter Estimation ... 105

C.3.1.8 Final Thought ... 106

C.3.2 Shaker Algorithm ... 106

C.3.2.1 Non-ocean net fisheries with no terminal stocks ... 106

C.3.2.2 Non-ocean net fisheries with at least one terminal stock ... 107

C.3.2.3 Ocean net fisheries with no terminal stocks ... 108

C.3.2.4 Ocean net fisheries with at least one terminal stock ... 108

C.3.3 Results of multi-phase catch ceiling algorithm test. ... 110

C.4 Maturation Process ... 115

C.4.1 The Biological Process 116
C.4.2 Mathematical Modeling Problem 116
C.4.3 New Main Engine 116
C.4.4 Cohort Objects and Data Tracking 117
C.4.5 Relationship to State Space Model 117
C.5 Production Processes 118
C.5.1 Pre-Spawning Mortality And Production Functions 118
C.5.2 Variable Truncated Ricker Function 119
C.5.3 Adult Equivalent Factors In Production Functions 121
C.6 Migration Process 122
C.6.1 Synthesizing Commonly Used Migration Algorithms 122
C.6.1.1 Background 122
C.6.1.2 Notation 122
C.6.1.3 State Space Model 123
C.6.1.4 PSC Chinook Model 124
C.6.1.5 PSC Selective Fisheries Model 126
C.6.1.6 Proportional Migration (PM) Model 130
C.6.1.7 Fishery Resource Allocation Model (FRAM) 136
C.6.2 Beta Advection-Diffusion Model 136
C.6.2.1 Background 136
C.6.2.2 The Model 136
C.6.2.3 Model Properties 137
C.6.2.4 Discussion 137

List of Figures

Figure 1 Expected step size as a function of relative location (i.e., *Dist_Scalar*) and relative time (*Time_Scalar*). 139

Figure 2 Expected step size as a function of current location for five values of relative t. 139

Figure 3 "Attraction Location" as a function of relative time for several values of *Move_Alpha*......... 140

List of Tables

Table 1 Fisheries included in PSC Chinook Model 15

Table 2 Stocks included in CRiSP Harvest Model 16

Table 3 Managers and other global classes and objects 71

Table 4 Available ProductionFunction derived classes 85

Table 5 Percent change in ceilinged fishery catches by year between PSC and NMFS catch ceiling algorithms. The four multi-phase ceiling fisheries (7, 8, 9, 20) are listed at the far right side of the table. 112

Table 6 Percent change in non-ceilinged fishery catches by year between PSC and NMFS catch ceiling algorithms. 113

Table 7 Percent change in stock escapements by year between PSC and NMFS catch ceiling algorithms (fisheries 1-15). 114

Table 8 Common notation used in this report (also see Figure 1). 122

Table 9 Dispersion parameters by week for the PSC Selective Fishery Model. 128

Table 10 Dispersion and non-dispersion (= 1 - dispersion) rate parameters for the South Puget Sound stock during week 40 used in the PSC Selective Fishery Model. 130

Table 11 Expected Step Size for values of relative location and time (*Move_Alpha* = 8.735; ? = 3.0).138

Table 12 "Attraction Location" by Relative Time and *Move_Alpha*. 140

1 Introduction

1.1 Background

The National Marine Fisheries Service (NMFS) has responsibility for administering the US Endangered Species Act (ESA) for anadromous fish and shared responsibility for managing ocean salmon fisheries. Because of this dual responsibility, NMFS has a need to evaluate the constraints of ESA considerations on harvest management, and the impacts of harvest and habitat management on the recovery of depleted stocks. These evaluations need to be made on a coastwide basis and should be internally consistent and consistent with models used in other management arenas.

In 1997 NMFS contracted with the University of Washington School of Fisheries to begin developing a single broad modeling framework that will assist NMFS in meeting its salmon management responsibilities. A Model Committee, composed of State, Tribal, and Federal salmon biologists and modelers, was formed to help develop model specifications, identify research priorities, ensure consistency with existing models, and certify research models for management use. Initial meetings with the Review Committee resulted in the following overall objectives for this modeling project:

1. Provide a common framework for both conservation risk assessment and harvest management analysis;
2. Expand the geographic scope of current harvest models;
3. Link coho and chinook salmon models;
4. Incorporate life cycle (production) models for both species to evaluate long-term harvest and conservation strategies;
5. Allow flexibility to accommodate new methods and model designs;
6. Provide an interface with a large subset of ocean and freshwater databases maintained by the Pacific States Marine Fisheries Commission.

Objective six was postponed because it was deemed beyond the scope of this project. During 1999 the Pacific Salmon Commission contracted with ESSA Technologies Ltd (Vancouver, BC) and UW to develop several enhancements to its Chinook Model, including an interface to databases.

1.2 Code Framework Overview

The Coast Model is not a single "model," but instead is a code framework that can be configured to represent many different models depending upon the specific process algorithms and data specified in the input files. At the present time (December 1999) the Coast Model only contains algorithms used by the Pacific Salmon Commission (PSC) chinook model. Section 1.5 gives an overview of the PSC chinook model; Section 2 describes its algorithms in detail. To configure the Coast Model to represent other salmon harvest models, such as the Fishery Regulation Analysis Model (FRAM) or the Proportional Migration (PM) model, new algorithms must be added to the processes. The general idea is to build up a library of algorithms for each process so the same code framework can be configured to represent a wide variety of models.

A key feature of the code is the clear separation of the fishing mortality and migration processes. All current salmon harvest models simulate changes in regulations by scaling fishery harvest rates up or down compared to some base period. Thus, it is difficult to simulate harvest regimes that differ significantly from the base period, especially in terms of changing the timing and location of individual fisheries. In contrast, the Coast Model performs three basic functions:

1. Divides time and space into any number of timesteps and regions;
2. Creates a biological system composed of any number and types of stocks that suffer natural mortality, mature, spawn, and migrate discretely through the time/space grid;
3. Superimposes a harvesting process over the biological system.

The Coast Model code framework is flexible because it uses an object-oriented programming language (C++) that encapsulates algorithms representing model processes (e.g., natural mortality, fishing mortality, maturation, migration, spawning) and associated data into single code objects. Thus, the new framework removes the need to require that all stocks use the same algorithms and data types, or the need for complicated branching code to treat stocks differently (e.g., if Stock A, do it this way; if Stock B, do it another way; if Stock C, do it a third way; etc.).

1.3 Specific Code Features

1.3.1 Discrete Time Chronology

The Coast Model is a discrete time model that loops over years and timesteps within each year. The user can specify any number of years and timesteps. The following processes occur within each timestep (in this order):

- Cohort Ageing
- Natural Mortality
- Fishing Mortality
- Maturation
- Spawning
- Migration

The above processes were needed to simulate the PSC Chinook Model. Additional processes could be added as more complicated algorithms are developed. For example, processes to update the physical characteristics of each geographic region (e.g., average sea surface temperature, average surface current direction and speed) and/or the biological characteristics (e.g., average abundance of prey species, average abundance of predator species) could be included at the start of each timestep. All following processes could have algorithms that use these region characteristics (e.g., natural mortality could depend on sea surface temperature and predator abundance; migration could depend on surface current direction and speed).

1.3.2 Cohort Based

The fundamental biological unit in the model is a cohort (i.e., a group of fish having the same biological characteristics). Cohorts can be defined by species, brood year, sex, growth group, maturation status, mark status, or tag status. Note the inclusion of maturation status. Immature and mature fish from the same brood year have different migration patterns and thus are treated as separate cohorts. As described later, the Maturation Process creates new mature cohorts.

1.3.3 Processes and Data Controlled by "Managers"

A "Manager" controls each process within a timestep. Each manager stores individual process objects and controls computation flow, but does not dictate specific process algorithms. For example, the SpawningManager stores a spawning process for each stock in each year and knows at which timestep(s) and region(s) to activate the processes. When it is time to perform spawning, the SpawningManager loops through all the stocks, passes the stock its correct spawning process, and directs the stock to implement the

process. Each spawning process contains all the data and algorithms necessary for a stock to perform spawning (i.e., generate one or more new cohorts of fish). Thus, some stocks might use a Ricker function, others might use a Beverton-Holt function, and still others might use a truncated linear function representing hatchery production.

1.3.4 Data Access by Generic Array

One of the greatest limitations of conventional model code is the fixed dimensionality of parameters. For example, a natural mortality rate might be indexed by age; or stock and age; or by stock, age, and location. Whenever the dimensionality changes, considerable re-coding is required. In the Coast Model, parameters are dimensioned at run time via "generic arrays." The first line of each data input file specifies the dimensionality of the parameter.

1.3.5 Data Request Manager

Process algorithms often require intermediate variables (e.g., sum of catches by fishery over a base period). To facilitate computing and storing intermediate variables, the Coast Model uses a Data Request Manager. At the end of each process, the manager checks to see if there are any data requests that require some action. If yes, the data are gathered and stored.

1.3.6 Input/Output Using Existing Tools

The Coast Model uses a "token-based" data input system. Each datafile contains key words, or tokens, that specify the type of data expected next. When duplicating the PSC chinook model, a utility program translates data from existing files into the token-based format. Output from the Coast Model is formatted into a single stream of Standardized Data Sentences (e.g., cohort abundance sentence, natural mortality sentence). Instead of creating formatted reports directly from the program, utility programs must be written to generate reports from the standard data sentences. Section 4 Output Language describes the output sentences in detail.

1.3.7 Multi-Timestep Iteration Capability

An "Iteration Manager" allows the code to:

- stop at the end of a timestep to evaluate certain conditions (e.g., the total catch in a fishery over several previous timesteps compared to catch quota);
- restart the model at a previous timestep using different values for control variables (e.g., fishing effort levels);
- repeat the above steps until the condition is satisfied.

1.4 Code Limitations

At the present time (December 1999), the Coast Model code has the following limitations:

- all processes are deterministic (i.e., there is no stochastic capability);
- current algorithms are limited to those used in the PSC Chinook Model;
- parameter estimation techniques have not been perfected (especially for the migration process and region specific harvest rates).

1.5 Pacific Salmon Commission Chinook Model

1.5.1 General Description of the PSC Chinook Model

The PSC Chinook Model was developed by the PSC Chinook Technical Committee to examine alternative management approaches to implement the PSC chinook rebuilding program (the next section contains a brief history of the model). The model is capable of simulating a large number of years, stocks (hatchery and natural), and fisheries (troll, net, and sport). (See Table 1 and Table 2.) A key feature of the model is the interaction between stocks through annual catch ceilings imposed upon fisheries that harvest multiple stocks. As stocks rebuild or decline at different rates over time, relative harvest rates in ceilinged fisheries also change. Single stock models cannot simulate this type of interaction.

Simulations are divided into two time periods: (1) a calibration period; and (2) a management simulation, or projection, period. The calibration period runs from 1979 through the last year for which model parameters can be estimated (usually one year earlier than the current year). The simulation period runs from the current year to any future year (usually about 10-15 years in the future). The PSC Chinook Model produces information to help evaluate the effects of changes in brood year survival rates and several management actions:

- pre-recruitment (i.e., age one) survival projections
- pre-spawning survival (i.e., inter-dam losses)
- enhancement activities
- catch ceilings (catch quotas)
- harvest rate strategies
- size limits.

Production parameters for both hatchery and natural stocks are estimated from historical data. Ocean survival rates for ages one through five are assumed fixed (at 0.5, 0.6, 0.7, 0.8, and 0.9, respectively) for all stocks. Survival rates to age one (also called Environmental Variability, or “EV,” scalars) are estimated during the calibration process. Other parameters are estimated by a technique known as “cohort analysis” or “virtual population analysis.” This type of analysis involves reconstructing an annual series of abundance estimates using catch and escapement data and making assumptions about natural and incidental mortalities. Once each cohort has been reconstructed, the following parameters are estimated:

- Cohort size for each age class at the beginning of each year
- Age specific harvest rates for each fishery
- Maturity schedule for all ages
- Estimates of incidental fishing mortalities.

The PSC chinook model is calibrated by finding a suite of stock and year-specific smolt to age one survival rates (EV scalars) that results in model outputs that most closely match user specific terminal run sizes, escapements, or catches for individual stocks during the base period. The user specifies the EV scalars for the simulation period, often taken to be the average of the base period values. The model results are known to be sensitive to the selection of the EV scalars for the simulation period.

Management changes are evaluated by changing key parameters, such as future catch ceilings or harvest rates, and rerunning the model.

Table 1 Fisheries included in PSC Chinook Model

Number	Fisheries	Abbreviation
1	Alaska Troll	Alaska T
2	Northern B.C. Troll	North T
3	Central B.C. Troll	Centr T
4	West Coast Vancouver Island Troll	WCVI T
5	Washington/Oregon Troll	WA/OR T
6	Strait of Georgia Troll	Geo St T
7	Alaska Net	Alaska N
8	Northern B.C. Net	North N
9	Central B.C. Net	Centr N
10	West Coast Vancouver Island Net	WCVI N
11	Juan de Fuca Net	J De F N
12	North Puget Sound Net	PgtNth N
13	South Puget Sound Net	PgtSth N
14	Washington Coast Net	Wash Cst N
15	Columbia River Net	Col R N
16	Johnstone Strait Net	John St N
17	Fraser River Net	Fraser N
18	Alaska Sport	Alaska S
19	North/Central B.C. Sport	Nor/Cen S
20	West Coast Vancouver Island Sport	WCVI S
21	Washington Ocean Sport	Wash Ocn S
22	North Puget Sound Sport	PgtNth S
23	South Puget Sound Sport	PgtSth S
24	Strait of Georgia Sport	Geo St S
25	Columbia River Sport	Col R S

Table 2 Stocks included in CRiSP Harvest Model

Number	Stocks	Abbreviation
1	Alaska South SE	AKS
2	Northern/Central B.C.	NTH
3	Fraser River Early	FRE
4	Fraser River Late	FRL
5	West Coast Vancouver Island Hatchery	RBH
6	West Coast Vancouver Island Natural	RBT
7	Upper Strait of Georgia	GSQ
8	Lower Strait of Georgia Natural	GST
9	Lower Strait of Georgia Hatchery	GSH
10	Nooksack River Fall	NKF
11	Puget Sound Fingerling	PSF
12	Puget Sound Natural Fingerling	PSN
13	Puget Sound Yearling	PSY
14	Nooksack River Spring	NKS
15	Skagit River Wild	SKG
16	Stillaguamish River Wild	STL
17	Snohomish River Wild	SNO
18	Washington Coastal Hatchery	WCH
19	Columbia River Upriver Brights	URB
20	Spring Creek Hatchery	SPR
21	Lower Bonneville Hatchery	BON
22	Fall Cowlitz River Hatchery	CWF
23	Lewis River Wild	LRW
24	Willamette River	WSH
25	Spring Cowlitz Hatchery	CWS
26	Columbia River Summers	SUM
27	Oregon Coastal	ORC
28	Washington Coastal Wild	WCN
29	Snake River Wild Fall	LYF
30	Mid Columbia River Brights	MCB

1.5.2 - Brief History of the PSC Chinook Model

During the negotiations which led to the Pacific Salmon Treaty in 1985, efforts to reach agreement on chinook management focused on strategies which would rebuild depressed natural stocks within an agreed-upon time period. At the technical level, several microcomputer models were developed to provide a method of consistently and objectively analyzing alternative options under consideration during the negotiations.

The computer models were designed to analyze how various combinations of fisheries management actions would affect rebuilding. Prior to the development of the models, information on the production levels for natural chinook stocks was often limited to measurements of catch and escapement in or near the

corresponding river of origin. Direct estimates of a significant component of overall production (i.e., harvest levels in ocean and near-shore mixed stock fisheries) were often not available for the natural stocks of interest. By integrating chinook life history assumptions with coded-wire-tag (CWT) recovery data, the models permitted the simulation of ocean and terminal harvest and escapement patterns.

The models simulated the process of rebuilding under hypothetical fishery policies that reduced harvest rates over time. As spawning escapements of depressed stocks increased to optimum levels, production increased. By maintaining fishery regimes, such as harvest ceilings, as run sizes progressively increased, rebuilding accelerated.

The models were initially designed to evaluate alternative fishery management regimes with respect to their implications for successfully rebuilding depressed chinook stocks by 1998. They progressed from simple cohort analyses designed to evaluate overall harvest rates and patterns of exploitation for single stocks or groups of stocks, to a "Multiple Stock Model" which incorporated multiple fisheries, stocks and brood years as well as stock- recruitment production functions. Intermediate steps included a simple "Forward Cohort Analysis" and a "Single Stock" multiple brood and fishery model (also including the stock-recruitment function).

While the "Single Stock" model achieved the goal of providing a set of mutually acceptable rules for evaluating proposals under consideration when the Pacific Salmon Treaty was being negotiated, it did not adequately represent results expected when several stocks were involved. Under the single stock approach, the progressive reductions in harvest rates in fisheries with ceilings resulting from increasing stock size over the course of the rebuilding cycle are transferred entirely to the single stock in the Model. In reality, the harvest rate changes in pre-terminal fisheries would be influenced by the abundance of the aggregate of stocks available. However, while the abundance of depressed components of the aggregate would be expected to increase as a result of increased escapement, the abundance of many components would remain relatively stable. As a result, the single stock approach would tend to underestimate the time required for rebuilding; it would present an overly optimistic picture of the effects of future reductions in harvest rates resulting from increased production.

Application of the Model to describe these mechanisms requires the assumption that proportional changes in total model fishery catch are represented by the actual changes in the real world catch. It also assumes that the stock composition in the Model catch reflects the relative contribution of these stocks to the actual catch (the abundance of unrepresented stocks is assumed to be constant).

If these assumptions are not met, the ceiling or quota mechanism on rebuilding will produce incorrect rebuilding schedules. The quota or ceiling mechanism will take effect at different harvest levels for each particular stock depending on the abundance of other stocks in the catch. For example, the rate at which a particular stock rebuilds may be accelerated by the presence of other stocks in the ceiling fisheries. If these other stocks respond to management measures at a faster rate, their abundance is increased and the relative contribution of the stock of interest to the fishery is reduced. This effect is similar to that resulting from enhancement where the increased abundance of hatchery fish will "saturate" the fishery under a fixed harvest ceiling and dilute the impact on wild stocks resulting in increased savings of wild fish to escapement.

More detailed stratification of fisheries was required to respond to a number of policy questions that were raised over time. The resolution needed for modeling may vary from issue to issue, depending upon the questions to be addressed and the availability of necessary data. The final Model used for the Pacific Salmon Treaty negotiations in 1984 incorporated four stocks and nine fisheries. The Model was modified in 1987 to enable it to simulate up to 25 fisheries and 26 stocks. In 1993 and 1994 the number of stocks was increased to 29 and 30, respectively.

By 1987, the effects of incidental mortality losses to the chinook rebuilding program had increasingly become a matter of concern as management agencies implemented various changes to fishing regulations to increase benefits under the fishery regimes established through the Pacific Salmon Commission. The

Model has been modified to more realistically reflect incidental mortality losses and permit the evaluation of regulations such as non-retention restrictions and size limit changes.

The Model was recoded into Microsoft QuickBasic™ language beginning in 1986 and was revised in a number of important ways to better meet needs under implementation of the Pacific Salmon Treaty.

The listing of the Snake River Fall Chinook stock as "endangered" under the US Endangered Species Act generated interest in harvest management decisions from stakeholders outside the normal harvest management "family." In 1993 the University of Washington School of Fisheries, with funding from the Bonneville Power Administration, began creating a user-friendly version of the PSC Chinook Model. The goal was to create a tool that both scientists and the general public could use to explore the effects of various harvest management regulations on chinook stock rebuilding.

The new user-friendly model, called CRiSP Harvest, was initially created in C++ under the UNIX operating system and was completed in 1995. In 1996 a PC version was developed to make the model more accessible to the general public. At the April 1996 Sustainable Fisheries Conference held in Victoria, British Columbia, Canada a new modeling approach was presented in which the harvest and migration processes were separated by using a State Space Model (Newman 1998). Discussions at that conference lead to the project to develop the Coast Model.

2 Coast Model Processes

2.1 Computation Flow

The text of Chapter 2 which follows discusses a number of Coast Model objects. The actual names of the objects as they appear in the source code are reflected here in bold type, as in **IterationManager** or **FisheryQuotaPolicy**. As a general rule the name of the object (or class) is the same as the base portion of both the .cpp file and .h file containing the code which implements that class, as in *"IterationManager.cpp"*. The text below references actual methods from the Coast Model in bold type followed by parenthesis, as in **maxAge()** (from the **Stock** class). See Section 5.2 for more information about the Coast Model source code naming conventions.

2.1.1 Overview

The Coast Model is a discrete time model that loops over years and timesteps within each year (see Section 5.6.2 for a description of main computation engine). Any number of years and timesteps can be specified by the user, but each year has the same number of timesteps. Any number of regions can be specified, but the number of regions must be the same in all years and timesteps. The following code segment from the "***config.data***" input file illustrates how time and space are controlled by the user.

```
# Configuration data for the NMFS Coast Model.

Configuration
  StartYear       1979
  EndYear         1999
  TimeSteps       4
  Regions         4
```

The following processes occur within each timestep (in this order):

- Cohort Ageing
- Natural Mortality
- Fishing Mortality
- Maturation
- Spawning
- Migration

Process algorithms loop over regions, stocks, ages, and fisheries, but not always in that order. Some process algorithms, such as fishing mortalities, may span multiple timesteps and require some iteration (i.e., making computations over the same timesteps multiple times with different values for some control variables). For example, prototype 10 in Appendix Section B.3.3 has three fisheries that have catch quotas over different time steps. An **IterationManager** is activated at the beginning and end of each timestep to control computation flow during iteration routines. See Section 5.5.2.5.2 (**IterationControl**), Section 5.5.2.5.3 (**IterationManager**), and Section 5.6.2 (The Engine) for more details.

2.1.2 PSC Chinook Model Implementation

Life cycle computations in the PSC chinook model are performed on an annual basis. There are no timesteps within years. The sequence of computations reverses the procedures employed in the cohort analysis used to generate the stock-specific input data. The annual computational sequence is outlined below:

- Population ageing
- Natural ocean mortality
- Preterminal (ocean) fishing mortality
- Maturation
- Terminal (nearshore and river) fishing mortality
- Pre-spawning mortality (inter-dam losses) for some stocks
- Production of progeny in the next year.

A modified version of the PSC chinook model (CRiSP Harvest) divided the terminal fishing mortality phase into separate nearshore and river harvest periods, thus giving three harvesting periods. Thus, within each timestep the PSC chinook model and its derivative have two instances of natural mortality (ocean and pre-spawning) and three instances of fishing mortality, whereas the Coast Model only has one natural mortality and fishing process per timestep.

The maturation process effectively creates a new mature cohort from each stock/age cohort, called the "Terminal Run." There is no specific migration algorithm in the PSC chinook model. Instead, there is an implied migration of mature fish from the ocean to the spawning region, because the terminal fishing mortality, pre-spawning mortality, and production processes only affect the terminal runs.

To cast the PSC chinook model in the Coast Model framework we used four timesteps and four regions (ocean, nearshore, river, and spawning). The table below shows which processes were activated in each of the four timesteps (indicated by an X).

Coast Process	**T1**	**T2**	**T3**	**T4**
Ageing	X			
Natural Mortality	X			X (IDLs)
Fishing Mortality	X (ocean)	X (nearshore)	X (river)	
Maturation	X			
Spawning				X
Migration	X	X	X	X

Migration is the only process activated in all timesteps and is used to move mature fish from the ocean to the nearshore, river, and spawning regions (see Section 2.7.2 for more details).

2.1.3 Future Processes

During the design stage for the Coast Model we envisioned three additional processes. The first would update the physical environment of each region (e.g., sea surface temperature, surface currents); the second would update the non-salmonid biological environment of each region (e.g., prey abundance, predator abundance). These two processes would be the first to occur within each timestep. These regional environmental parameters would then be available for other processes. The third additional process would be a growth process to update the average length of fish within a cohort. This process would occur after ageing and before natural mortality.

2.2 Cohort Ageing

2.2.1 Overview

Each cohort maintains an **Age** property (see Section 5.5.2.1 for more details about **Cohort** objects). Cohort age in years is defined to be the current year minus the brood year (i.e., the year in which spawning occurred). The *"config.data"* file lists all cohorts needed to seed the model (sample code below). New cohorts created by the spawning process are assigned age 0 (see Section 5.5.3.2.2 for more details about generating new **Cohort** objects). The ageing process occurs first during each timestep, and is currently activated only during the first timestep of each year.

It is not necessary that each cohort have the same number of age classes. In fact, each **Stock** has three additional age-related properties: **maxAge()**, **FirstHarvestAge**, and **AdultAge**.

2.2.2 PSC Chinook Model Implementation

In the PSC chinook model configuration of the Coast Model spring stocks have six age classes, whereas fall stocks have five age classes (see Appendix Section C.2 for a complete discussion about spring and fall stock ageing in the PSC chinook model). Here is a sample from the "***config.data"*** file for simulating the 9812 version of the PSC chinook model:

```
# Stock names and abbreviations.
  StockName          "Alaska South SE"
    StockNumber       1
    StockAbbreviation AKS
    Run               Spring
    ProductionType    Wild
    MaxAge            6
    FirstHarvestAge   3
    AdultAge          4
  end Stock
…
  StockName          "Fraser Early"
    StockNumber       3
    StockAbbreviation FRE
    Run               Fall
    ProductionType    Wild
    MaxAge            5
    FirstHarvestAge   2
    AdultAge          3
  end Stock
```

Below is an example from the **Val98.coh** input file that initializes cohorts for the 9812 version of the PSC chinook model (see Section 3.10 for further code input language details). Each record lists the brood year, starting abundance, and the region in which to place the abundance. In this case, all abundances are placed in the ocean region (1). Note that six cohorts are initialized for spring stocks and five for fall stocks.

```
# Initial cohorts by stock and brood year.
# Brood year, initial abundance, initial region.

Cohorts
  Stock "Alaska South SE"  # Initial cohorts for stock number: 1
    Cohort     1973  3345.02050867604  1
    end Cohort
```

```
    Cohort     1974  20295.0016210456  1
    end Cohort
    Cohort     1975  44085.8599317261  1
    end Cohort
    Cohort     1976  99300.8052578326  1
    end Cohort
    Cohort     1977  296962.592912911  1
    end Cohort
    Cohort     1978  379046.727021314  1
    end Cohort
  end Stock
…
  Stock "Fraser Early"  # Initial cohorts for stock number: 3
    Cohort     1974  31588.8350783806  1
    end Cohort
    Cohort     1975  152651.339455101  1
    end Cohort
    Cohort     1976  205339.159233051  1
    end Cohort
    Cohort     1977  324139.988558743  1
    end Cohort
    Cohort     1978  697510.038476283  1
    end Cohort
  end Stock
```

2.2.3 Future Ageing Processes

Future versions of the Coast Model could maintain age at higher resolution (e.g., 4.3 years) by having ageing algorithms that update age at every timestep. Higher resolution ages may then be used for other processes (e.g., harvest algorithm that require fish size as a function of age).

2.3 Natural Mortality

2.3.1 Overview

The Coast Model has only one natural mortality process:

$$NatMort = N \cdot r$$

where N = the cohort abundance at the start of the natural mortality process and r is the natural mortality rate. This is defined in the Coast Model code in **NaturalMortalityManager::takeNaturalMortality()**.

2.3.2 PSC Chinook Model Implementation

The PSC chinook model has two types of natural mortality:

- Natural mortality for all stocks in the ocean
- Natural mortality for stocks migrating up the Columbia and Snake Rivers

Natural mortality in the ocean is age specific and occurs only during the first timestep. Here is a code segment from the **Val98.nat** input data file. Note that age 1 cohorts from spring stocks do not suffer any

natural mortality while in the river. The basic idea of this model is that all stocks suffer the same natural ocean mortality based on their "ocean age" (i.e., number of years spent in the ocean).

```
NatMortRateData StockXyearXageXtimeXregion

TimeStep 1
  Region 1
    Stock 1 # Alaska South SE // Run type = Spring
      Age 1 NaturalMortality 0.0
      Age 2 NaturalMortality 0.5
      Age 3 NaturalMortality 0.4
      Age 4 NaturalMortality 0.3
      Age 5 NaturalMortality 0.2
      Age 6 NaturalMortality 0.1
    end Stock
…
    Stock 3 # Fraser Early // Run type = Fall
      Age 1 NaturalMortality 0.5
      Age 2 NaturalMortality 0.4
      Age 3 NaturalMortality 0.3
      Age 4 NaturalMortality 0.2
      Age 5 NaturalMortality 0.1
    end Stock
```

Three stocks suffer mortality while migrating up the Columbia and Snake Rivers. The mortality rates are stock, year, and age specific. The PSC chinook model refers to these mortalities as "Inter-Dam Losses" or IDLs. These are simulated in the Coast Model by assigning natural mortalities in the last timestep (4) and region (4). Below is a code segment from the **Val98.nat** input data file assigning mortalities to the upriver bright (URB) stock.

```
TimeStep 4
  Region 4
    Stock URB # Fall stock.
      Age    1 NaturalMortality 0.0
      Ages 2:5
        Year 1979 NaturalMortality 0.0774
        Year 1980 NaturalMortality 0.4479
        Year 1981 NaturalMortality 0.5152
        Year 1982 NaturalMortality 0.4627
        Year 1983 NaturalMortality 0.1098
        Year 1984 NaturalMortality 0.0352
… etc
```

2.3.3 Future Natural Mortality Processes

The current framework allows natural mortalities to be stock, age, year, timestep, and region specific. While this allows considerable flexibility in assigning natural mortalities, future users may wish to use more complicated algorithms. For example, the natural mortality rate could be a function of predator and/or prey abundance within a region, or it might depend on river temperatures and flow. The code framework

can be modified to accommodate these more complicated algorithm by defining new natural mortality methods.

The current Coast Model framework does not support using instantaneous total mortality rates (i.e., natural plus fishing). See Appendix Section C.3 for a discussion of how natural and fishing mortality rates can be combined into a single "mortality process."

2.4 Fishing Mortality

2.4.1 Overview

Computing fishing mortalities is the most complex portion of the Coast Model (see Section 5.5.3.1 for complete details). Our goal was to create a flexible code framework that would allow scientists and modelers to add new algorithms with a minimum of coding effort. The specific algorithms currently supported in the Coast Model are based on those from the PSC chinook model. The actions taken during the harvest process are the following:

1. Generate temporary data storage for the harvest mortality data. (This results in the creation of **FisheryUnit** and **HvMort** objects.)
2. Instruct the **PolicyControlManager** to perform **preHarvestManagement()**.
3. For each Fishery:
    - Compute legal catches.
    - Compute shaker incidental mortality.
    - Compute CNR incidental mortality.
4. Instruct the **PolicyControlManager** to perform **postHarvestManagement()**.
5. Update all cohort abundances by applying the computed harvest mortalities from each Fishery.

Note that the organization of the harvest process presumes that within a timestep harvests for individual fisheries may be computed independently. It also presumes that legal catches and incidental mortalities within the same fishery/region/timestep can be computed independently. The only way to achieve interactive effects across fisheries is by using the Iteration feature of the model (see Section 5.5.2.5). Similarly, it is further assumed that harvest in a specific fishery and region (in a particular timestep) may be computed independently from other fisheries and/or regions. It is possible that this last requirement may be loosened at some point in the future, but for the moment the code structure adopts this as an invariant.

The discrete nature of the Coast Model harvest process may be a significant limitation. Appendix Section C.3 provides a detailed discussion of the harvest process limitations in the Coast Model and discusses possible code changes to remove these limitations. With a moderate amount of re-coding the Coast Model harvest process could be expanded to use instantaneous mortality rate equations which would allow full interaction between fisheries operating within the same region and timestep. This would require combining the natural mortality and fishing mortality processes into a single mortality process.

2.4.2 Legal Catches

2.4.2.1 SingleCohort

In the Coast Model, the **HarvestProcess** object contains the information necessary to compute catches by a single fishery of legal sized fish from a cohort within a given year, timestep, and region. Only one algorithm is currently supported, as follows:

$$LegalCatch = N \cdot PV \cdot HR \cdot FP \cdot EffortScalar$$

where *N* is the regional cohort abundance (at the start of the fishing mortality process), *PV* is the proportion of the regional abundance that is vulnerable to the gear (e.g., due to size limits), *HR* is the "base period" harvest rate, *FP* is a "fishery policy" input scalar to adjust the base period harvest rate for the current year, timestep, and region, and *EffortScalar* is an additional policy control variable that can be adjusted during iteration routines. *PV*s, *HR*s, and *FP*s are defined via separate input data files and can have almost any dimensionality using the generic array data storage system. Thus, this simple linear harvest equation combined with flexible parameter dimensionality provides an extremely robust tool for computing legal catches. This equation is actually in the **LinearHarvestProcess::computeCatch()** method, which (contrary to the usual naming convention) is found in file ***"HarvestProcess.cpp"***.

2.4.2.2 Multiple Cohorts (Safe-Guarded Secant Algorithm)

Some harvest policies require that the catch of all cohorts within a single year, timestep, and fishery are equal to a given number. If the single cohort algorithm described above is used for all **HarvestProcesses**, then the *EffortScalars* for each process can all be adjusted by the same amount such that the total catch of all cohorts equals the desired amount. The catch ceiling algorithm of the PSC chinook model operates in this fashion. If any **HarvestProcesses** are non-linear, then this methodology will not work.

The safe-guarded secant algorithm solves for the effort level in a fishery during a given timestep and region such that the total catch in that fishery hits a specific catch quota (ceiling), *regardless of the types of equations used to compute the cohort specific catches*. One nice feature of this algorithm is that when all the catch equations are linear (as is the case for the PSC Chinook Model), then the safe-guarded secant method solves in a single iteration, just as the PSC Chinook Model algorithm does. The following brief description of the safe-guarded secant algorithm is from Jan 19, 1999 minutes.

Since each **HarvestProcess** object can have a different type of algorithm to compute legal catch at the cohort level, a **FisheryPolicy** object that has a catch quota objective requires a general algorithm that will adjust the effort level in a fishery to meet the quota. Mathematically, we have the following problem:

Find E such that

$$C = F(E) = K$$

where

- C = total catch in a fishery;
- E = relative effort level in the fishery;
- K = desired catch quota;
- F(E) = unknown function.

The unknown function F(E) is the sum of all the unknown **HarvestProcess** functions for each cohort. The problem can be restated as follows:

Find E such that

$$G(E) = F(E) - K = 0.$$

The code used in the Coast Model is similar to the secant method used in Function RTSEC in Numerical Recipes. For further algorithm details see the code and comments in the Coast Model. The **MultiCeilingIterationConrol** class implements this functionality. Also see section 5.5.2.5 for a description of the code implementing the harvest iteration system.

2.4.3 Incidental Mortalities

Virtually all "real world" fishing processes kill more fish than are landed legally. Some fish encounter the fishing gear and suffer mortality, but are never brought on board the vessel. For example, in salmon fisheries some fish hooked by commercial troll and sport fishermen or gilled by net fishermen escape the gear before it is retrieved. A portion of these fish may die from their encounter with the gear. These types of mortalities are referred to as "drop off" mortalities. Since these fish are never even seen by scientific observers on board the vessel, drop off mortalities are difficult to study and estimate.

Fish brought on board the vessel but not landed are generally referred to as "bycatch." For example, many fisheries have size limits. Any captured fish whose length is below the size limit must be released. In salmon hook and line fisheries, these undersized fish are referred to as "shakers" because they are "shaken" off the gear. Some of the shakers survive, but others die due to the stress of being captured and released. The shaker mortality rate (i.e., the fraction of shakers that die) is gear dependent. Troll and sport gears cause relatively low shaker mortality, since the fish are captured individually and in many cases can be released without serious injury. Net fisheries cause higher shaker mortalities, because the capture process is more stressful.

Another type of salmon bycatch occurs when it is illegal to keep some species of salmon. For example, it may be legal to keep all species of salmon except chinook. Incidental mortalities caused by releasing prohibited species are called "non-retention" mortalities. Note that non-retention mortalities include both legal and sub-legal sized fish. The Coast Model supports several PSC chinook model algorithms for computing "chinook non-retention," or "CNR," mortalities.

In the Coast Model the **HarvestManager** maintains a generic array of **Shaker** and **CNR** objects. Each of these objects contains all the data and algorithms for computing incidental mortalities and is specified by the user via the ***.shk** and ***.cnr** input files. If the user does not want the model to compute incidental mortalities, these two files can be omitted. The following sections briefly describe the algorithms.

2.4.3.1 Shakers

As noted above, the legal catches are computed first and stored by fishery, stock, and age during each timestep and region. Legal catches are computed for each fishery independently. Thus, during one loop through the fisheries, the catches of one fishery in a timestep and region are not affected by catches of other fisheries operating in that same timestep and region.

Shakers must be computed before the CNRs. In the 1995 version of PSC chinook model the only mortality rate parameter required was a "*ShakerMortRate*" that was gear specific (Troll = 0.30; Net = 0.90; Sport = 0.30). In the 1998 version there are three fishery specific incidental mortality rates:

- *SublegalShakerMortRate*;
- *LegalShakerMortRate*;
- *DropOffRate*.

Shakers are fish that are killed incidentally while harvesting legal size fish. These include "DropOffs" (sublegal and legal size fish that drop off the gear before being brought to the boat) and "Releases" (sublegal size fish that are brought to the boat, but are released because they are below the legal size limit).

The PSC Chinook Model shaker algorithm was difficult to implement in the Coast Model because it relied upon a subjective concept (preterminal vs terminal; ocean net fishery) of which stocks were considered vulnerable to each fishery during a timestep. The new Coast Model shaker algorithm resolves this dilemma by including a "vulnerability" table in each **Shaker** object for which not all stocks are vulnerable. Since shakers are computed for each fishery independently, each fishery/region/timestep has a **Shaker** object.

Appendix Section C.3.2 contains a detailed discussion (including examples) of the shaker algorithm and how vulnerability tables are created for various situations in the PSC chinook model. The following sections describe how shaker mortalities are computed for the stocks that are considered vulnerable to a fishery. Thus, keep in mind that all references to stocks in the following equations refer only to stocks that are vulnerable to the referenced fishery.

The shaker equations shown below are implemented in the classes derived from the base **Shaker** class. **PSCShaker** contains most of the code while other subclasses contain further specializations. See Section 5.5.3.1.6.1 for a list of the various **Shaker** subclasses.

2.4.3.1.1 Compute the "*StockWts*"

The relative contribution of each stock in each fishery (called the "stock weight") is computed by:

$$StkWgt_{s,f} = \frac{FP_{s,f} \cdot \sum_{a} Catch_{s,a,f}}{\sum_{s} FP_{s,f} \sum_{a} Catch_{s,a,f}}$$

Note that the numerator is the catch of stock *s* by fishery *f* and denominator is the total catch by fishery *f*. Note also that if all catches by fishery *f* are multiplied by a common scaling factor, the stock weight term is unchanged.

2.4.3.1.2 Compute the "*TotalPNV*" and "*TotalPV*"

These variables represent the total number of sublegal and legal fish recruited to the gear in fishery *f*.

$$TotPNV_{f} = \sum_{s} \sum_{a} N_{s,a} \cdot PNV_{a,f} \cdot StkWgt_{s,f}$$

$$TotPV_{f} = \sum_{s} \sum_{a} N_{s,a} \cdot PV_{a,f} \cdot StkWgt_{s,f}$$

2.4.3.1.3 Compute the "*EncounterRate*"

The encounter rate for each fishery is computed by

$$EncRte_{f} = \frac{TotPNV_{f}}{TotPV_{f}}$$

2.4.3.1.4 Compute the "*FracNV*"

$$FracNV_{s,a,f} = \frac{StkWgt_{s,f} \cdot N_{s,a} \cdot PNV_{a,f}}{TotPNV_{f}}$$

2.4.3.1.5 Compute "TotalShakers"

Total shakers in fishery f is the product of the total catch by fishery f times the encounter rate times the shaker mortality rate. Note that if all the catches in a given fishery are multiplied by a common scaling factor, total shakers is also multiplied by that factor.

$$TotShak_f = ShakMortRte_f \cdot EncRte_f \cdot \sum_f FP_{s,f} \cdot \sum_a Catch_{s,a,f}$$

where

$$ShakMortRte_f = SublegalShakMortRte_f + DropOffRate_f$$

2.4.3.1.6 Distribute "*TotalShakers*" by stock and age using the *FracNVs*

Total shakers are distributed by stock and age to get sublegal shakers. These sublegal shakers are stored separately (from the legal drop offs) for possible use in subsequent CNR computations.

$$SublegalShak_{f,s,a} = TotShak_f \cdot FracNF_{s,a}$$

LegalDropOffs are added to the sublegal shakers to get total shakers:

$$TotShak_{f,s,a} = SublegalShak_{f,s,a} + LegalDropOffs_{f,s,a}$$

where

$$LegalDropOffs_{f,s,a} = LegalCatch_{f,s,a} \cdot DropOffRate_f$$

Since the *LegalShakerMortRate* parameter is not used to calculate shakers, the Coast model does not include that parameter with the input specifications for shakers.

[Programming Note: The PSC chinook model QuickBasic code computes the encounter rate and the *FracNVs* in Sub **CalcEncRte** (this sub appears to be identical in CTC95, CTC98, and CTC99). Sub **CalcShaker** is called at the end of Sub **CalcEncRte** to compute the total shakers and distribute them among the cohorts. Sub **CalcShaker** also computes the CNR mortalities.]

2.4.3.2 Chinook Non-Retention Mortalities

Chinook Non-Retention (CNR) mortalities are sublegal and legal size fish killed in fisheries targeting on other salmon species. In CTC98 the computation of these mortalities proceeds in three steps:

- Compute CNR ratios (e.g., legal CNR morts to legal catch);
- Compute CNR mortalities without considering multiple encounters (e.g., apply CNR ratio to legal catch);
- Adjust CNR mortalities for multiple encounters (e.g., multiply CNR morts by a multiple encounter adjustment scalar).

2.4.3.2.1 Compute the ratios relating legal CNR mortalities to the legal catch and sublegal CNR mortalities to the shakers.

This is the same idea as in CTC95; the only difference is the mortality rate used. Three methods can be used to compute these ratios. The naming of the methods shown below reflects the naming of the

subclasses deriving from **CNR** which implement each of the equations. See also Section 5.5.3.1.6.2 for a full list of the **CNR** subclasses.

CNR_HarvestRatio Method

If the current year relative effort level is between zero and one (i.e., $0 < RelHR < 1$), then compute the ratios as follows:

$$CNRSublegalRatio_f = CNRSublegalSel_f \cdot \frac{1 - RelHR_f}{RelHR_f}$$

$$CNRLegalRatio_f = CNRLegalSel_f \cdot LegalCNRMortRate_f \cdot \frac{1 - RelHR_f}{RelHR_f}$$

where

$$LegalCNRMortRate_f = LegalShakerMortRate_f + DropOffRate_f$$

CNR_SeasonLength Method

This method is similar to the ratio method, except the relative effort level is determined by the season lengths. Note that the *DropOffRate* is not used in this method.

$$CNRSublegalRatio_f = CNRSublegalSel_f \cdot \frac{CNRSeasonLength_f}{LegalSeasonLength_f}$$

$$CNRLegalRatio_f = CNRLegalSel_f \cdot LegalShakerMortRate_f \cdot \frac{CNRSeasonLength_f}{LegalSeasonLength_f}$$

Possible QB Program Bug. There is an inconsistency in the 1998 QuickBasic code that needs to be resolved. The inconsistency is in the 1999 code as well. For fisheries that do not have incidental mortality rate changes, the above formula for calculating the *CNRLegalRatio* uses only the *LegalShakerMortRate*. On the other hand, fisheries that do have incidental mortality rate changes use the sum of the *LegalShakerMortRate* and the *DropOffRate*. Adding the *DropOffRate* would be consistent with the HarvestRatio method. The QuickBasic code is below, with the important lines in bold:

```
CASE 0   '....RT

      IF RT > 0 AND RT < 1

        TmpR = (1 - RT) /

        Tmp4 = CNRSelect(0, CNRIndx%) * TmpR                  '.....

        '..... CNRSelect is the selectivity factors

        'compute legal IM+ dropoff and check for changes 2/9/98

        TmpSM = ShakerMortRte(Fish%, 2) + ShakerMortRte(Fish%, 3)

        FOR ICheck% = 1 TO NumIMChange%
```

```
          TestFish% = IMChange!(ICheck%, 1)

          TestYr% = IMChange!(ICheck%, 2)

          IF TestFish% = Fish% AND Yr% >= TestYr% - 79 THEN

            TmpSM = IMChange!(ICheck%, 4) + IMChange!(ICheck%, 5)

          END IF

        NEXT ICheck%

        '.....

        Tmp5 = CNRSelect(1, CNRIndx%) * TmpSM * TmpR              '..... Legals

      END IF


    CASE 1   '....Season Length

      Tmp6 = CNRData(Yr%, CNRIndx%, 1) / CNRData(Yr%, CNRIndx%, 0)  '..... CNR/Regular
Season

      Tmp4 = CNRSelect(0, CNRIndx%) * Tmp6                           '..... SubLegal

      TmpSM = ShakerMortRte(Fish%, 2)


[?? TmpSM = ShakerMortRte(Fish%, 2) + ShakerMortRte(Fish%, 3) ??]


      '......Now check for changes in IM rates and substitute if needed 2/9/98

      FOR ICheck% = 1 TO NumIMChange%

        TestFish% = IMChange!(ICheck%, 1)

        TestYr% = IMChange!(ICheck%, 2)

        IF TestFish% = Fish% AND Yr% >= TestYr% - 79 THEN

          TmpSM = IMChange!(ICheck%, 4) + IMChange!(ICheck%, 5)

        END IF

      NEXT ICheck%
```

Thus, it appears that the equation above should be:

$$CNRLegalRatio_f = CNRLegalSel_f \cdot LegalCNRMortRate_f \cdot \frac{CNRSeasonLength_f}{LegalSeasonLength_f}$$

where

$$LegalCNRMortRate_f = LegalShakerMortRate_f + DropOffRate_f$$

CNR_ReportedEncounter Method

This method uses reported encounters to determine the relative effort levels.

$$CNRSublegalRatio_f = CNRMortRate_f \cdot \frac{1}{EncRate_f} \cdot \frac{RptSublegalEncounters_f}{RptLegalCatch_f}$$

[Programming Note: The QuickBasic code does not use 1/*EncRate*. Instead it effectively back-calculates the encounter rate from *TotalCatch* and *SublegalShakers* (before *DropOffs* are added).]

$$CNRLegalRatio_f = CNRMortRate_f \cdot \frac{RptLegalEncounters_f}{RptLegalCatch_f}$$

where

$$CNRMortRate_f = SublegalShakerMortRate_f + DropOffRate_f$$

[Question: Should the *CNRMortRate* be the same for both the sublegals and legals? Seems like the rate use for the legal CNRs should be the sum of the *LegalShakerMortRate* plus the *DropOffRate*.]

2.4.3.2.2 Compute legal and sublegal CNR mortalities without considering multiple encounters

Note that for the sublegals, the ratio is applied only to the sublegal shakers, not the sublegal shakers plus the drop offs.

$$SublegalCNRMorts_{f,s,a} = CNRSublegalRatio_f \cdot SublegalShakers_{f,s,a}$$

$$LegalCNRMorts_{f,s,a} = CNRLegalRatio_f \cdot LegalCatch_{f,s,a}$$

If the Reported Encounter method was used in Step 1, then no further adjustments are necessary and the CNR computations are finished. The **CNR_HarvestRatioMultipleEncounter** and **CNR_SeasonLengthMultipleEncounter** methods require an adjustment for multiple encounters.

2.4.3.2.3 Adjust CNR mortalities for multiple encounters

If appropriate, adjust the CNR mortalities by computing a *CNRScalar* adjustment factor (“appropriateness” is explained later). All methods start by computing the number of potential encounter periods during the base period, as follows:

$$BaseEncounterPeriods_f = \frac{BasePeriodSeason_f}{RecaptureInterval_f}$$

[Programming Note: We can compute this with the utility and pass it in as data from the cnr file.]

The following computations are done for each cohort (i.e., fishery, stock, age).

CNR_HarvestRatioMultiple Encounter Method

This method is applied only if two conditions are satisfied. First, the base period exploitation rate for this cohort must be greater than zero (i.e., this cohort must be harvested by this fishery). Second, the scalar used to adjust the cohort specific catches by this fishery to meet its ceiling ($RelHR_f$) is between zero and one (i.e., the CNR season must have increased relative to the base period).

First, convert the discrete-time base period exploitation rate to an instantaneous rate (scaled to the number of base period encounter periods).

$$InstBaseHR_{f,s,a} = \frac{-\ln(1 - BaseHR_{f,s,a})}{BaseEncounterPeriods_f}$$

This base period instantaneous rate is then adjusted to reflect the new amount of effort or season length. This factor is then used to compute the scalar. The adjustment factor is computed by

$$Adjust = (1 - RelHR_f) \cdot BaseEncounterPeriods_f$$

A new temporary instantaneous rate is given by

$$TempRate = Adjust \cdot InstBaseHR_{f,s,a}$$

Use this temporary rate is used to back-calculate a new discrete exploitation rate:

$$ER_{f,s,a} = 1 - e^{-TempRate}$$

If this value is > 1, the sublegal and legal CNR scalars are both given by:

$$CNRScalar_{f,s,a} = \frac{ER_{f,s,a}}{BaseHR_{f,s,a} \cdot (1 - RelHR_f)}$$

The sublegal and legal CNR mortalities computed in step 2 are multiplied by this scalar value to get the final CNR mortalities.

CNR_SeasonLengthMultipleEncounter Method

Computations are slightly different for sublegal and legal size fish. Both methods use an adjustment factor to reflect changes in season length.

$$Adjust = \frac{CNRSeasonLength_f}{CNRSeasonLength_f + LegalSeasonLength_f}$$

Note that *Adjust* is always less than zero.

Sublegal Computations

An adjusted base harvest rate is computed by

$$AdjBaseHR_{f,s,a} = Adjust \cdot BaseHR_{f,s,a} \cdot SublegalSel_f$$

If this adjusted base harvest rate is zero, then the sublegal scalar is set to zero. If it is greater than zero, a new instantaneous rate is computed as

$$TempRate = -\ln(1 - AdjBaseHR_{f,s,a}) \cdot LegalShakerMortRate \cdot Adjust$$

This temporary rate is used to back-calculate a new discrete exploitation rate:

$$ER_{f,s,a} = 1 - e^{-TempRate}$$

The sublegal CNR scalar is

$$SublegalCNRScalar_{f,s,a} = \frac{ER_{f,s,a}}{AdjBaseHR_{f,s,a} \cdot LegalShakerMortRate}$$

The sublegal CNR mortalities computed in Step 2 are multiplied by this scalar to get the final sublegal CNR mortalities.

Legal Computations

An adjusted base harvest rate is computed by

$$AdjBaseHR_{f,s,a} = Adjust \cdot BaseHR_{f,s,a} \cdot LegalSel_f$$

If this adjusted base harvest rate is zero, then the legal CNR scalar is set to zero. If it is greater than zero (but not = 1), a new instantaneous rate is computed as

$$TempRate = -\ln(1 - AdjBaseHR_{f,s,a}) \cdot LegalShakerMortRate \cdot Adjust$$

This temporary rate is used to back-calculate a new discrete exploitation rate:

$$ER_{f,s,a} = 1 - e^{-TempRate}$$

The legal CNR scalar is

$$LegalCNRScalar_{f,s,a} = \frac{ER_{f,s,a}}{AdjBaseHR_{f,s,a} \cdot LegalShakerMortRate}$$

The legal CNR mortalities computed in Step 2 are multiplied by this scalar to get the final legal CNR mortalities.

Possible Program Bug. Once the legal and sublegal CNR scalars are computed, the program only applies the scalars if the *Adjust* term is greater than one. However, as noted above, this term is always less than one. Thus, the scalars are never applied. The ultimate effect is that multiple encounters are never actually accounted for whenever the season length method is used. To verify this, we inserted debugging code within the program to print out the maximum values for the *Adjust* term and the two scalars (sublegal and legal) whenever the season length method is used. The code and results are listed below.

```
        ' so if adjust >1 then did multiple encounters

        IF Adjust > 1 THEN

          '..... Estimate CNR mortality losses

          CNRShakCat(LocF, CNRIndx%, Stk%, Age%) = Tmp * Tmp4 * SSub

          CNRLegal(LocF, CNRIndx%, Stk%, Age%) = MDLCatch!(LocF, Fish%, Stk%, Age%) * Tmp5 * SLeg

        ELSE

          '..... Estimate CNR mortality losses

          '... recall that Tmp has TotShak (hence IM rate) already in it

          CNRShakCat(LocF, CNRIndx%, Stk%, Age%) = Tmp * Tmp4

          CNRLegal(LocF, CNRIndx%, Stk%, Age%) = MDLCatch!(LocF, Fish%, Stk%, Age%) * Tmp5

        END IF


' Norris CNR debugger.

    IF CNRMeth%(Yr%, CNRIndx%) = 1 THEN

        PRINT #20, Yr%, Fish%, Stk%, Age%, Adjust, SSub, SLeg

    END IF

' End debugger.



      END IF

    END IF

  NEXT Age%

NEXT Stk%

EXIT SUB
```

2.5 Maturation

2.5.1 Overview

The Coast Model supports only one maturation algorithm—a simple rate model. When the **MaturationManager** invokes the **maturateCohorts()** method, the following steps occur:

- Loop through all cohorts;
- If the cohort is immature, loop through all regional abundances for that cohort;
- Apply the appropriate maturation rate and decrement the regional abundance;

$$MatureFish = N \cdot r$$

$$N^{'} = N - MatureFish$$

where N is the starting (i.e., pre-maturation process) regional abundance, N' is the ending (i.e., post-maturation process) regional abundance, and r is the maturation rate.

- If a corresponding mature cohort (i.e., same **CohortID** (see ***"Cohort.cpp"***) except for maturation status) already exists within this region, add the new mature fish to the existing mature cohort.
- If a corresponding mature cohort does not exist within this region, create one and add the new mature fish to the new mature cohort.

2.5.2 PSC Chinook Model Implementation

In the PSC chinook model maturation only occurs during the first timestep and the ocean region (1). All stocks have age specific maturation rates, and some stocks have the same maturation rates every year. Below is a code sample from the **Val98.mrt** input file assigning fixed maturation rates. Note that the oldest age cohorts always have a 1.0 maturation rate (age 6 for spring stocks and age 5 for fall stocks).

```
MaturationData StockXyearXageXtimeXregion

# No maturation occurs during timesteps 2:4.
TimeSteps 2:4 MatRate 0.0

# Maturation only occurs at the end of the first timestep.
TimeStep 1
  Regions 2:4 MatRate 0.0
  Region 1  # Preterminal region.
    Age 1 MatRate 0.0

    # Data for stocks with fixed maturation rates for all years.
    # Years 1979:1999

    Stock NTH # All years; run type = Spring
      Age 2 MatRate 0.0
      Age 3 MatRate 0.051396523
      Age 4 MatRate 0.14471728
      Age 5 MatRate 0.69004977
      Age 6 MatRate 0.99999988
    end Stock
```

```
    Stock FRE # All years; run type = Fall
      Age 2 MatRate 0.026498009
      Age 3 MatRate 0.1446432
      Age 4 MatRate 0.68596339
      Age 5 MatRate 1
```

Below is a code sample from the **Val98.mrt** input file assigning maturation rates that vary by year.

```
# Data for stocks with variable maturation rates.
    Stock AKS
      Year 1979 # Stock AKS; run type = Spring
        Age 2 MatRate 0.0
        Age 3 MatRate 0.0576
        Age 4 MatRate 0.1212
        Age 5 MatRate 0.6487
        Age 6 MatRate 0.99999994
      end Year
      Year 1980 # Stock AKS; run type = Spring
        Age 2 MatRate 0.0
        Age 3 MatRate 0.0576
        Age 4 MatRate 0.1212
        Age 5 MatRate 0.6487
        Age 6 MatRate 0.99999994
      end Year
      Year 1981 # Stock AKS; run type = Spring
        Age 2 MatRate 0.0
        Age 3 MatRate 0.0045
        Age 4 MatRate 0.1212
        Age 5 MatRate 0.6487
        Age 6 MatRate 0.99999994
      end Year
```

2.5.3 Future Maturation Processes

Future versions of the Coast Model could have more complicated maturation algorithms. For example, the maturation rate could be dependent on the physical characteristics of the region (e.g., water temperature) or the average individual size of the cohort (e.g., larger fish could have a higher maturation rate).

2.6 Spawning

2.6.1 Overview

The **SpawningManager** controls when spawning will occur by maintaining a **ProductionTable** (generic array) of **Production** objects (see ***"SpawningManager.h***" and Section 5.5.3.2 for more details). A **Production** object contains all the information necessary to compute the number of new age 0 fish produced by the mature adult spawners from a given stock. At each timestep, every stock and region are examined. If there is a **ProductionTable** entry for that stock and region in the current timestep, then production will occur. The total adult mature regional abundance for that stock and region (regional escapement) is calculated, and the **spawn()** method of the appropriate Production object is invoked. The output is the new abundance of age 0 fish for that stock and region.

Note that the above code structure gives the Coast Model the flexibility to allow spawning in any region or timestep. This is accomplished by assigning **Production** objects to regions and timesteps via the ***.prd**

input file. The **CohortGenerator** always places the new age 0 cohorts in the region in which they spawn (see Section 5.5.3.2.2 for more details). Users are cautioned that they must properly assign age-specific transition matrices in the ***.tm** input file to move age 0 cohorts from their spawning region at the desired time.

Each **Production** object maintains a list of **ProductionFunctions** (see Section 5.5.3.2.1), each of which describes the relationship between *a* number of mature adult spawners from a given stock and the new age 0 fish they will produce. Note the use of the indefinite article "*a*" instead of the definite article "*the*" before the phrase "number of adult spawners." This is to emphasize that a **ProductionFunction** does not necessarily describe the relationship between the *total* adult spawners and their progeny. We do this because many salmon production algorithms involve several functions.

Consider the case of a hatchery stock for which the hatchery has a limited capacity, say 5,000 adult spawners, and any excess escapement is allowed to spawn in the wild. If the total escapement is 7,500 in a given year, production from the first 5,000 spawners would use one function (e.g., a linear function) and the remaining 2,500 spawners would use a different function (e.g., a Ricker function).

All **ProductionFunctions** have minimum and maximum spawner values to define which portion of the total escapement is to be used for each function (input as the first and second parameters in the input file; see Section 3 for complete input code specifications). The age 0 fish produced by each function are added together to get total production. In the above example, the linear function would use 5,000 spawners and the Ricker function would use 2,500 spawners. Mathematically, the rule for determining the number of adult spawners passed to a **ProductionFunction** (as a function of total adult escapement) is given by:

$$AdultSpawners = \begin{cases} 0 & TotAdltEsc < min \\ TotAdltEsc - min & min \le TotAdltEsc < max \\ max & max \le TotAdltEsc \end{cases}$$

All **ProductionFunctions** also have a parameter called the Environmental Variability (EV) Scalar (input as the third parameter in the input file; see Section 3 for complete input code specifications). This parameter is used to account for known or predicted deviations around the deterministic functions.

The following code segment from the **Val98.prd** input file illustrates how **ProductionFunctions** are linked to create a **Production** object. Note that the minimum and maximum spawner values for each function occur in sequence and do not overlap. For example, in 1983 the first 5,318 spawners use the first linear function, the next 244 spawners (= 5,562 - 5,318) use the second linear function, the next 5,000 spawners (= 10, 318 – 5,318) use the Ricker function, and any additional escapement is ignored. Also note that the *EVScalar* values for all functions within the same year are equal, but between years they are different.

```
StockNum 9 #Production functions for stock: GSH
  Year 1979        #For stock: GSH
    Production Linear  0  5318  0.576309919  101.088866871763
    Production Ricker  5318  10318  0.576309919  2.813  72371.2428651556
0.181012304888616
  end Year
  Year 1980        #For stock: GSH
    Production Linear  0  5318  0.338252485  101.088866871763
    Production Ricker  5318  10318  0.338252485  2.813  72371.2428651556
0.181012304888616
  end Year
  Year 1981        #For stock: GSH
    Production Linear  0  5318  1.36516976  101.088866871763
    Production Ricker  5318  10318  1.36516976  2.813  72371.2428651556
0.181012304888616
```

```
  end Year
  Year 1982        #For stock: GSH
    Production Linear  0  5054.47415595914  1.3196733  101.088866871763
  end Year
  Year 1983        #For stock: GSH
    Production Linear  0  5318  0.560896635  101.088866871763
    Production Linear  5318  5562.0650328853  0.560896635  17.5666110352434
    Production Ricker  5562.0650328853  10562.0650328853  0.560896635  2.813
72371.2428651556  0.181012304888616
```

2.6.1.1 LinearProduction

A linear function is typically used for hatchery production and requires the following parameters:

- *min* (minimum number of adult spawners)
- *max* (maximum number of adult spawners)
- *EVScalar*
- *Slope*

The specific formula is a straight line through the origin (with the effective slope of the line equal to the product of the *Slope* and *EVScalar* parameters):

$$AgeZeroFish = AdultSpawners \cdot Slope \cdot EVScalar$$

If the hatchery production is less efficient beyond a given level of spawners, more than one **ProductionFunction** can be assigned to a **Production** object. Consider the example below for the RBH stock from the PSC chinook model (sample code from the **Val98.prd** input file). In 1979-1981 there is a single linear **ProductionFunction** with a maximum of 6,472 adult spawners, a slope of 250.6 (i.e., each spawner produces 250.6 age 0 fish deterministically), and EV Scalar values of 1.04, 0.77, and 0.44, respectively. In 1982 a second linear function is added. This second function is invoked only for adult escapement in excess of 6,472 up to a maximum of 8,971. This indicates that the hatchery capacity has increased to 8,971 adult spawners, but note that the slope is reduced from 250.6 to 94.1 signifying less efficient production. The table below shows the number of age 0 fish produced by 7,500 spawners given the example **ProductionFunctions** for the RBH stock.

Year	*TotAdltEsc*	*AdultSpawners*	EffectiveSlope	Age 0 Fish
1979	7,500	6,472	260.1	1,689,049
1980	7,500	6,472	192.8	1,247,22
1981	7,500	6,472	109.6	709,313
1982	7,500	6,472 (first function)	100.0	647,177
		1,028 (second function)	37.5	38,581
			Total 1982	685,758

```
ProductionFunctions StockXyearXtimeXregion  #Production functions by stock and
year.
TimeStep 4
Region 4

StockNum 5 #Production functions for stock: RBH
  Year 1979        #For stock: RBH
    Production Linear  0  6472  1.04126465  250.635577084189
  end Year
```

```
Year 1980        #For stock: RBH
  Production Linear  0  6472  0.769133806  250.635577084189
end Year
Year 1981        #For stock: RBH
  Production Linear  0  6472  0.4372769  250.635577084189
end Year
Year 1982        #For stock: RBH
  Production Linear  0  6472  0.398971677  250.635577084189
  Production Linear  6472  8970.96933755731  0.398971677  94.0663138467215
end Year
Year 1983        #For stock: RBH
  Production Linear  0  6472  0.0862767845  250.635577084189
  Production Linear  6472  9973.86853326432  0.0862767845  94.0663138467215
end Year
```

2.6.1.2 RickerProduction

This function is the familiar Ricker spawner/recruit function multiplied by the *EVScalar* parameter (to account for annual variability) and a parameter to convert predicted adult recruitment to age 1 fish (*RectAtAgeOne*). The required parameters are

- *min* (minimum number of adult spawners)
- *max* (maximum number of adult spawners)
- *EVScalar*
- $\boldsymbol{a}$ (Ricker A parameter)
- $\boldsymbol{b}$ (Ricker B parameter)
- *RectAtAgeOne*

The specific formula is:

$$AgeZeroFish = EVScalar \cdot RectAtAgeOne \cdot AdultSpawners \cdot e^{\boldsymbol{a} \cdot (1 - \frac{AdultSpawners}{\boldsymbol{b}})}$$

2.6.1.3 EnhancedRickerProduction

This function was developed to implement an algorithm in the PSC chinook model. It allows for enhancement of natural stocks (also called supplementation) in which a portion of the natural spawners are removed for hatchery production. The following parameters are required for this function:

- *min* (minimum number of adult spawners)
- *max* (maximum number of adult spawners)
- *EVScalar*
- $\boldsymbol{a}$ (Ricker A parameter)
- $\boldsymbol{b}$ (Ricker B parameter)
- *RectAtAgeOne*
- Density dependence flag
- *HatchProd* (hatchery productivity parameter)
- *SmoltSurvRt* (smolt survival rate to age one)
- *EnhProp* (maximum proportion of the wild stock that can be removed for hatchery production)

- *Smolts* (smolt production change over some base period)

The first step is to determine number of wild spawners that must be removed to meet the smolt production goal (*Smolts*). These are called enhancement spawners (*EnhSpawners*) and are computed as follows:

$$EnhSpawners = \frac{Smolts \cdot SmoltSurvRt}{e^{HatchProd}}$$

The number of enhancement spawners removed may not exceed a maximum allowable percentage of the adult spawners, called the *MaxBrood*:

$$MaxBrood = EnhProp \cdot AdultSpawners$$

EnhSpawners is truncated to *MaxBrood*, if necessary.

Smolts from hatchery production are returned back to the river of origin. If no competition between wild and hatchery smolts is assumed (i.e., density dependence flag is false), then natural and hatchery production are computed independently and added together. The naturally produced smolts are computed from the remaining natural spawners using the Ricker function

$$AgeZeroFish = EVScalar \cdot RectAtAgeOne \cdot WildSpawners \cdot e^{\alpha \cdot (1 - \frac{WildSpawners}{\beta})}$$

where $WildSpawners = AdultSpawners - EnhSpawners$. The hatchery produced smolts are computed using a linear function, as follows:

$$AgeZeroFish = AdultSpawners \cdot e^{EnhProd} \cdot EVScalar$$

When density dependence is assumed (i.e., density dependence flag is true), *AgeZeroFish* is computed using a Ricker curve, but the "effective" size of the spawning stock is increased to reflect the fact that eggs from some of the spawners are reared in a hatchery. The enhancement efficiency of the hatchery is given by

$$EnhEff = \frac{e^{EnhProd}}{e^{\alpha}}$$

In general, *HatchProd* is greater than α so *EnhEff* is usually greater than one. The effective number of spawners is given by

$$EffectiveSpawners = EnhSpawners \cdot \frac{EnhEff}{\alpha} + (AdltEsc - EnhSpawners)$$

Age 0 fish are then computed with the Ricker function using effective spawners.

$$AgeZeroFish = EVScalar \cdot RectAtAgeOne \cdot EffSpawners \cdot e^{\alpha \cdot (1 - \frac{WildSpawners}{\beta})}$$

2.6.1.4 VariableTruncationRickerProduction

This function was created to implement an algorithm from the PSC chinook model. However, it appears that there is a bug in this algorithm and we anticipate that it will be removed. For a complete discussion of this production function see Appendix Section C.5.2.

2.6.2 PSC Chinook Model Implementation

Production functions in the PSC chinook model use data from several files. For example, the 9812 version of the model uses the following files (production related parameters are in parens):

- 9812v.op6 (density dependence flag)
- Cal9807.bse (natural ocean mortality rates, α, optimum escapement, truncation flag)
- Enhanc98.enh (*HatchProd*, *SmoltSurvRt*, *EnhProp*, *Smolts*)
- Mat98.msc (stock and year specific maturation rate parameters)
- 9801.idl (natural pre-spawning mortality rates)
- 9812p.evo (*EVScalar*)

Our goal with the Coast Model was to consolidate all data required for **Production** objects into a single file. We wrote a utility program that converts data from CTC chinook model input files to a single Coast Model production file (***.prd**). The following CTC chinook model parameters were used in Coast Model **ProductionFunctions** without modification:

- Density dependence flag
- α
- *HatchProd*
- *SmoltSurvRt*
- *EnhProp*
- *Smolts*
- *EVScalar*

The utility program computer code used to convert PSC chinook model input files to Coast Model input files contains comment statements describing the algorithms. The methods are briefly described here.

The *min* and *max* spawners used in all Coast Model **ProductionFunctions** were computed from the natural ocean and pre-spawning mortality rates, maturation rates, optimum escapements, and truncation flags. These natural mortality and maturation rates also were used to compute the *RectAtAgeOne* Coast Model parameter. The *Slope* parameter in the Coast Model **LinearProduction** functions is given by e^{α} for hatchery stocks without enhancement and by $e^{HatchProd}$ for the second linear function used by hatchery stocks with enhancement. In the Ricker functions, the β parameter was computed as follows (Hilborn's approximation):

$$\beta = \frac{OptimumEscapement}{0.5 - 0.07 \cdot \alpha}$$

2.6.3 Future Spawning Processes

A limiting feature of the Coast Model is that the **SpawningManager** passes *total* adult escapement to all **ProductionFunctions**. Thus, the Coast Model spawning process tacitly assumes that spawners from all adult ages are equally productive. Each stock has a property called **AdultAge** which defines the youngest age to include in adult spawners. The **SpawningManager** computes the total adult spawners before passing that argument to the **Production** object. Thus, in its current form, the Coast Model cannot support any production function that requires age specific data. More generally, it cannot support any production function that requires partitioning total adult escapement into age, sex, size, or other components. Any future spawning processes that require such partitioning will require some code changes to the **SpawningManager** in addition to writing new **ProductionFunctions**.

2.7 Migration

2.7.1 Overview

The Coast Model migration process currently has only one algorithm, which is based on the transition matrix algorithm used in Ken Newman's State Space Model (SSM) (Newman 199?). All commonly used migration algorithms can be modeled using the SSM approach (see Appendix Section C.5 for a complete report on using the SSM approach to model other salmon model migration algorithms).

In matrix notation, the deterministic SSM consists of two equations:

$$\mathbf{n}_t = \mathbf{M}_t \mathbf{S}_t \mathbf{n}_{t-1} \qquad \text{(state process)}$$

$$\mathbf{c}_t = \mathbf{H}_t \mathbf{n}_t \qquad \text{(observation process)}$$

The abundance vectors $\mathbf{n}_t$ and $\mathbf{n}_{t-1}$ are composed of *R* elements (one abundance for each region). Each migration matrix $\mathbf{M}_t$ is an *R* x *R* square matrix of $m_{i,j}$ elements ($m_{i,j}$ = the fraction of the abundance in region *j* moving to region *I*). The elements in each column must sum to one. Each survival matrix $\mathbf{S}_t$ and each harvest matrix $\mathbf{H}_t$ is a diagonal matrix with *R* elements (e.g., $s_{r,t}$, $h_{r,t}$). In expanded form the state process equation looks like this:

$$\begin{bmatrix} n_{1,t} \\ \vdots \\ n_{R,t} \end{bmatrix} = \begin{bmatrix} m_{1,1} & \cdots & m_{1,R} \\ \vdots & & \vdots \\ m_{R,1} & \cdots & m_{R,R} \end{bmatrix} \begin{bmatrix} s_{1,t} & & \\ & \ddots & \\ & & s_{R,t} \end{bmatrix} \begin{bmatrix} n_{1,t-1} \\ \vdots \\ n_{R,t-1} \end{bmatrix}$$

In terms of the Coast Model processes, the product of the survival matrix $\mathbf{S}_t$ and the abundance vector $\mathbf{n}_{t-1}$ is simply an updated abundance vector after the natural mortality, fishing mortality, maturation, and spawning processes have occurred. Thus, we can define a new updated abundance vector by $\mathbf{n}'_{t-1} = \mathbf{S}_t \mathbf{n}_{t-1}$. Thus, the migration matrix can be thought of as being applied to the updated cohort abundance vector. Each element of the abundance vector after migration can be written

$$n_{r,t} = \sum_{j=1}^{R} m_{r,j} n'_{r,t-1} \ .$$

The migration process is controlled in the Coast Model by the **MigrationManager** and implemented in the **TransitionMatrix**.

2.7.2 PSC Chinook Model Application

The migration matrices used to simulate the 9812 version of the PSC chinook model are given below. Note the following:

- Separate matrices are given for fall and spring stocks to reflect the fact that spring stocks remain in the river for their first year of life (see Appendix Section C.2 for a complete discussion about spring and fall stock ageing in the PSC chinook model).
- Immature fall stocks of all ages do not migrate during timesteps 1 – 3. However, during the last timestep (4), the new cohorts produced by the spawning process (which precedes the migration process) are moved from the river region (4) to the ocean region (1).
- The spawning process places new cohorts into the river region. Thus, immature age 0 spring cohorts have the identity migration matrix to keep these cohorts in the river. At age 1 they are moved from the river to the ocean.
- Mature fall and spring stocks of all ages follow a "boxcar" style migration (i.e., moving from one region to the next at the end of each timestep until reaching the river region). During the last timestep (4), the fish located in the river region do not migrate anywhere to simulate killing the fish that have spawned.

```
TransitionMatrix RunXageXmaturityXtime

Run Fall
  Immature  # Immature cohorts.
    TimeSteps 1:3
      Data
        1.0  0.0  0.0  0.0
        0.0  1.0  0.0  0.0
        0.0  0.0  1.0  0.0
        0.0  0.0  0.0  1.0
    TimeStep 4
      Data
        1.0  0.0  0.0  1.0
        0.0  1.0  0.0  0.0
        0.0  0.0  1.0  0.0
        0.0  0.0  0.0  0.0
  end maturity

  Mature  # Mature cohorts.
    TimeSteps 1:3
      Data
        0.0  0.0  0.0  0.0
        1.0  0.0  0.0  0.0
        0.0  1.0  0.0  0.0
        0.0  0.0  1.0  1.0
    TimeStep 4
      Data
        0.0  0.0  0.0  0.0
        1.0  0.0  0.0  0.0
        0.0  1.0  0.0  0.0
        0.0  0.0  1.0  0.0
  end Maturity
end Run
Run Spring
  Immature  # Immature cohorts.
```

```
    Age 0
        Data
          1.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  1.0
    Age 1
        Data
          1.0  0.0  0.0  1.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  0.0
    Ages 2:6
        Data
          1.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  1.0
  end Maturity

  Mature  # Mature cohorts.
    TimeSteps 1:3
      Data
        0.0  0.0  0.0  0.0
        1.0  0.0  0.0  0.0
        0.0  1.0  0.0  0.0
        0.0  0.0  1.0  1.0
    TimeStep 4
      Data
        0.0  0.0  0.0  0.0
        1.0  0.0  0.0  0.0
        0.0  1.0  0.0  0.0
        0.0  0.0  1.0  0.0
  end Maturity
end Run
end transition matrix
```

3 Input Language

3.1 Introduction

The Coast model employs a hierarchical token-based language to process input data. The basic form is:

[*token*] [*data*]

All tokens and data are case sensitive. Whitespace is ignored, allowing tokens to span as many or as few lines as desired. Strings consisting of more than one word (such as fishery names or stock names) may be specified in quotation marks (“ ”).

For the purposes of this document [*int*] is used to represent a single data item which should be an integer value, and [*float*] is used to represent a single floating point value (either single or double precision is acceptable). In general, brackets [ ] are used in this document to denote some type of data item whose literal value is dependent upon the data. The brackets themselves do not appear in the input data files.

3.2 Token Types

Tokens fall into the general categories described below.

3.2.1 Simple Tokens

Simple tokens consist of a [*token*] [*data*] pair.

Example:

```
StartYear       1979
```

This sets the value of the StartYear variable to 1979.

3.2.2 Command Block Tokens

Command block tokens are those where data is another series of token/data units. A command block defines a new nested context wherein only selected tokens specific to that command block are recognized. The context is exited upon reading an “end” token, at which time the previous context is reactivated. The basic form for a command block token is

[*token*] [*data*] end

Example:

```
Configuration
    StartYear       1979
    EndYear         1999
  end
```

In this example, “Configuration” is a command block token. “StartYear” and “EndYear” are simple tokens within the “Configuration” context. Thus, they may also be described as “Configuration” subtokens (see section 3.2.4).

3.2.3 Generic Array Tokens

Generic array tokens are a special subset of command block tokens. All generic array tokens share a common set of tokens used to specify slices of the array data, as well as any tokens specific to the array used to specify the final parameter data. Generic array tokens are described further below.

3.2.4 Subtokens

Subtokens are tokens valid only in a particular context defined by a command block. As such, this is not really a special type of token. Simple tokens, command blocks, and generic array tokens may all be considered subtokens if they are specific to a particular containing command block.

3.2.5 Special Tokens

include [*filename*]	This token is valid in all contexts, and specifies that subsequent token/data pai will be read from the given *filename* until that file is exhausted, after which processing will continue with the next line in the current file. A typical top-lev data file will exist exclusively of "include" specifications, with each reference file containing a subset of the model data.
end [*optional comment*]	Specifies the end of a command block. This token has no [*data*] component. The remainder of the line following the "end" command is ignored, and may b used for comments.
End	Same as "end."
#	This symbol denotes a comment. The remainder of the line is ignored.

Example:

The current top-level file for the Coast model, "coast.data" consists of the following:

```
include Val98.bhr
include Val98.cei
include Val98.cnr
include Val98.coh
include Val98.fp
include Val98.fsh
include Val98.mrt
include Val98.nat
include Val98.pnv
include Val98.prd
include Val98.shk
include Val98.tm
```

3.3 Generic Arrays

Generic array tokens are command block tokens used to provide parameter data for a generic array. They take the following form:

[*generic array token*] [*generic array dimension specifier*] [*data*] end

where the data typically consists of a number of token/data pairs, including both *generic array subtokens* (described below) and simple tokens. The *generic array dimension specifier* describes the dimensionality of the array data about to be provided. Any of the available dimension specifiers may be used with any *generic array*

token, providing the user the flexibility of defining the dimensionality of the given parameter data at runtime. All available dimension specifiers and *generic array subtokens* are described below.

The [*data*] portion of a generic array command block applies to all the data in the array, unless specific slices are referenced through the use of *generic array subtokens,* which are special types of command block tokens.

Example:

```
MaturationData StockXyearXageXtimeXregion

  # No maturation occurs during timesteps 2:4.
  TimeSteps 2:4 MatRate 0.0
end MaturationData
```

In this example, "MaturationData" is the *generic array token* specifying a new context, which will contain maturation rate data. "StockXyearXageXtimeXregion" is the *generic array dimension specifier*, which defines 5 dimensional array to hold the maturation rate data, to be indexed by stock, year, age, time, and region. "TimeSteps" is a *generic array subtoken* specifying that upcoming data will be applied to the slice of the array corresponding to the time dimension. "2:4" is the data for the "TimeSteps" token, specifying the particular group of timesteps for the upcoming data. "MatRate" is a simple token and "0.0" is the data for that token. This data will be applied to all elements of the array specified by the current slice (in this case timesteps 2, 3, and 4). "end" is the end token for the "MaturationData" command block. The remainder of the line after the "end" token is ignored, so the final "MaturationData" is simply a comment for the reader.

The effect in this example is to set the maturation rate for all stocks, years, ages, and regions in timesteps 2, 3, and 4 to the value 0.0. Maturation rates for all other timesteps are left at the default value.

Data for a generic array is typically specified by nesting several layers of *generic array subtokens*. In this way the set of all the parameter data is successively narrowed until only the group of desired elements are being referenced, after which a simple token is provided to specify the data for those elements.

Example:

```
MaturationData StockXyearXageXtimeXregion

# No maturation occurs during timesteps 2:4.
TimeSteps 2:4 MatRate 0.0

# Maturation only occurs at the end of the first timestep.
TimeStep 1
  Regions 2:4 MatRate 0.0
  Region 1  # Preterminal region.
    Age 1 MatRate 0.0

    # Data for stocks with fixed maturation rates for all years.
     Stock NTH # All years; run type = Spring
      Age 2 MatRate 0.0
      Age 3 MatRate 0.051396523
      Age 4 MatRate 0.14471728
      Age 5 MatRate 0.69004977
      Age 6 MatRate 0.99999988
    end Stock
  end Region 1
end TimeStep 1
end MaturationData
```

This example builds upon the previous one by specifying further maturation rate data for timestep 1. *Generic array subtokens* shown here are "TimeSteps," "TimeStep," "Regions," "Region," "Stock," and "Age." Each describes a particular dimension of the generic array for which data is about to be provided. Starting at the top, the example specifies maturation rates as follows:

> 0.0 for timesteps 2-4 (all regions, stocks, ages, etc.);
>
> 0.0 for regions 2-4 in timestep1 (all stocks, ages, etc.);
>
> 0.0 for age1, region 1, timestep1;
>
> specific rates for ages 2-6, stock NTH, region 1, timestep 1.

Since the years are not specified, all of the data applies to all years.

3.3.1 Generic Array Subtokens

Generic array subtokens are command block tokens valid within the context of any *generic array token* command block. However, *generic array subtokens* behave differently than normal command block tokens in one respect. The "end" token, which normally closes a command block, is used at all levels in a generic array specification except for the innermost. The simple token which finally specifies data for some portion of the array subsumes the "end" token for the innermost block and renders it unnecessary. It is a syntax error to use an "end" token immediately following a simple token/data pair in a generic array specification.

Look again at the previous example. After the first maturation rate of 0.0 is given, no "end" is supplied for the enclosing "TimeSteps 2:4" block. That "end" is subsumed by the "MatRate" simple token, and the context reverts to the enclosing one, which references all elements of the array. Next the context is refined to timestep 1, then to regions 2-4. Another simple token is given specifying a rate of 0.0. This subsumes the "end" for the enclosing "Regions" block, and the context reverts to the most recent "TimeStep 1." This is then refined to region 1, then again to age 1, at which another final simple token appears, providing a maturation rate for that context (0.0 for all age 1, region 1, timestep1 entries). No "end" is supplied for that "Age 1" context, and the most recent context is again reverted to. At this point the context is region 1, timestep 1 (all stocks, ages, and years are implicit, since they have not been refined in the current context). A stock refinement is given, followed by a number of age refinements. For each age, a simple "MatRate" token subsumes the "end" which would normally appear with a command block. After the "MatRate" specification for "Age 6" the context reverts to the enclosing level, which is "Stock NTH." The writer wished to close that level without specifying any further maturation rates. Thus, the "end" token appears to close out that level of refinement (the word "Stock" after the token "end" is just a comment for the reader). Similarly, the enclosing refinements of Region and TimeStep are closed before finally supplying the "end" token for the entire generic array command block. Note that the contexts created by the *generic array subtoken* command blocks are nested, and thus represent an ordering. The successive "end" tokens each exit the most recent enclosing block. All "end" tokens for all nesting levels are required, with the exception of the innermost level, which includes the final simple token specifying actual parameter data, as discussed here.

The following is a list of all *generic array subtokens*, valid in the context of any *generic array token.* Note that data of the form *x:y* represents a range of values where *x*<=*y*.

Token	Data	Meaning
Year	[*int*]	new year context
Years	[*int:int*]	new year range context
TimeStep	[*int*]	new timestep context
TimeSteps	[*int:int*]	new timestep range context
Region	[*int*]	new region context
Regions	[*int:int*]	new region range context
Stock	[*stockname*]	new stock context
Stock	[*stockabbrev*]	new stock context
Stock	[*int*]	new stock context
Stocks	[*int:int*]	new stock range context
Fishery	[*fisheryname*]	new fishery context
Fishery	[*fisheryabbrev*]	new fishery context
Fishery	[*int*]	new fishery context
Fisheries	[*int:int*]	new fishery range context
Age	[*int*]	new age context
Ages	[*int:int*]	new age range context
Run	[*runtype*]	new run type context. Valid run types are “Spring” and “Fall”

3.3.2 Generic Array Dimension Specifiers

The following is a list of all *generic array dimension specifiers*. Each is used to specify the dimension of a generic array as shown in Section 3.3. Any *generic array dimension specifier* may be used with any *generic array token*. This list will be expanded as needs require.

StockXageXtime

StockXyear

StockXyearXtimeXregion

StockXyearXageXtime

StockXyearXageXtimeXregion

StockXyearXageXtimeXmaturity

XFishery

FisheryXyear

FisheryXregionXtimeXstockXage

FisheryXregionXtimeXstockXageXmark

FisheryXyearXage

FisheryXyearXstock

FisheryXyearXstockXage

FisheryXyearXrunXage

MaturityXtime

RunXageXmaturityXtime

3.4 Top Level Tokens

The following tokens are valid at the top level of the hierarchy. The term "*gen array dim spec*" is used as an abbreviation for "*generic array dimension specifier*."

Token	Data	Meaning
Configuration	[*command block data*] end	new configuration context
HarvestRateData	[*gen array dim spec*] [*data*] end	harvest rate data
CeilingScalars	[*command block data*] end	new ceiling scalar context
MultiTimeStepCeilingScalars	[*command block data*] end	new multiceiling context
CeilingData	[*command block data*] end	new ceiling context
CNRData	[*gen array dim spec*] [*data*] end	chinook non-retention
Cohorts	[*command block data*] end	new cohort context
FPData	[*gen array dim spec*] [*data*] end	fp (fishery policy) data
FisherySchedule	[*command block data*] end	new fishery schedule context
MaturationData	[*gen array dim spec*] [*data*] end	maturation rate data
NatMortRateData	[*gen array dim spec*] [*data*] end	natural mortality data
PnvData	[*gen array dim spec*] [*data*] end	percent non-vulnerable
ProductionFunctions	[*gen array dim spec*] [*data*] end	production data
ShakerData	[*gen array dim spec*] [*data*] end	shaker data
TransitionMatrix	[*gen array dim spec*] [*data*] end	transition matrices

3.5 Configuration Tokens

The "Configuration" command block token opens a context where all global configuration information for the model simulation is specified. The following tokens are valid within that context:

Token	Data	Meaning
StartYear	[*int*]	calendar year of simulation start
EndYear	[*int*]	calendar year of simulation end
TimeSteps	[*int*]	simulation timesteps per year
Regions	[*int*]	number of regions
FisheryName	[*fishery name*]	
	[*command block data*] end	new fishery configuration context
StockName	[*stock name*]	
	[*command block data*] end	new stock configuration context

3.5.1 Fishery Configuration Tokens

The "FisheryName" token opens a new fishery configuration context for the specification of basic fishery information. The following tokens are valid within that context:

Token	Data	Meaning
FisheryNumber	[*int*]	integer reference number
GearType	[*gear type*]	valid gear types are "troll," "net," and "sport"
FisheryAbbreviation	[*abbrv*]	abbreviation for fishery

3.6 Stock Configuration Tokens

The "StockName" token opens a new stock configuration context for the specification of basic stock information. The following tokens are valid within that context:

Token	Data	Meaning
StockNumber	[*int*]	integer reference number
Run	[*run type*]	valid run types are "Fall" and "Spring"
ProductionType	[*prod type*]	valid prod types are "Wild" and "Hatchery"
StockAbbreviation	[*abbrv*]	abbreviation for this stock
MaxAge	[*int*]	maximum stock age
FirstHarvestAge	[*int*]	youngest age harvested for this stock
AdultAge	[*int*]	adult age for this stock

Example:

```
Configuration
  StartYear        1979
  EndYear          1999
  TimeSteps        4
  Regions          4

# Fishery names and gear types.
  FisheryName     "Alaska T"
    FisheryNumber 1
    GearType       troll
  end Fishery

# Stock names and abbreviations.
  StockName          "Alaska South SE"
    StockNumber       1
    StockAbbreviation AKS
    Run               Spring
    ProductionType    Wild
    MaxAge            6
    FirstHarvestAge   3
    AdultAge          4
  end Stock
end Configuration
```

3.7 Harvest Rate Tokens

The token "HarvestRateData" is a generic array token which enters a new context for providing harvest rate data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
BaseHR	[*float*]	base harvest rate

3.8 Ceiling Tokens

There are three ways to enter ceiling data. The "CeilingScalars," "MultiTimeStepCeilingScalars," and "CeilingData" command block token each open new contexts.

3.8.1 CeilingScalars

The "CeilingScalars" token opens a context for the input of fishery ceiling scalar data. The scalar parameters given are used to scale average catches over the base period to set a quota for the desired fishery in each region and timestep for the specified year. Valid tokens in this context are:

Token	Data	Meaning
Fishery	[*fishery name*] [*command block data*] end	new fishery context
BasePeriodStart	[*int*]	calendar year of start of base period
BasePeriodEnd	[*int*]	calendar year of end of base period

Within the context opened by the "Fishery" command block the following tokens are valid:

Token	Data	Meaning
Year	[*int*] [*forceflag*] [*float*]	these fields specify the ceiling year, force flag, and scalar value, respectively. Valid values for *forceflag* are "forced" and "unforced."

Example:

```
CeilingScalars
  BasePeriodStart 1979
  BasePeriodEnd   1984
  Fishery "Alaska T"                # Fishery number 1
    Year 1985   forced  0.7787260906481610 #catch
    Year 1986   forced  0.8500192082294420 #catch
    Year 1999 unforced  0.7135184337829930 #ceiling control
  end Fishery
  Fishery "North T"                # Fishery number 2
    Year 1985   forced  1.1421682619914400 #catch
  end Fishery
end CeilingScalars
```

3.8.2 MultiTimeStepCeilingScalars

The "MultiTimeStepCeilingScalars" token opens a context for the input of fishery ceiling scalar data that enforces constant effort scalars over multiple timesteps. The scalar parameters given are used to scale average catches over the base period to set a quota for the desired fishery in the specified year. Valid tokens in this context are:

Token	Data	Meaning
Fishery	[*fishery name*] [*command block data*] end	new fishery context
BasePeriodStart	[*int*]	calendar year of start of base period
BasePeriodEnd	[*int*]	calendar year of end of base period
FirstTimeStep	[*int*]	first time step of ceiling management each year
LastTimeStep	[*int*]	last time step of ceiling management each year

Within the context opened by the "Fishery" command block the following tokens are valid:

Token	Data	Meaning
Year	[*int*] [*forceflag*] [*float*]	these fields specify the ceiling year, force flag, and scalar value respectively. Valid values for *forceflag* are "forced" and "unforced."

Example:

```
MultiTimeStepCeilingScalars
  BasePeriodStart 1979
  BasePeriodEnd   1984
  FirstTimeStep   1
  LastTimeStep    2
  Fishery "Alaska N"                # Fishery number 7
    Year 1985   forced  1.2161450300168500 #catch
    Year 1986   forced  0.7290534608357260 #catch
  end Fishery
end MultiTimeStepCeilingScalars
```

3.8.3 CeilingData

Ceiling data may also be entered directly, without the use of base period scalars. The "CeilingData" token is used to enter this context. It functions as a *generic array token*, except that no *generic array dimension specifier* is used, and valid tokens are restricted to only certain *generic array subtokens*.

Valid tokens in this context are:

Token	Data	Meaning
Year	[*int*]	new year context
Years	[*int:int*]	new year range context
TimeStep	[*int*]	new timestep context
TimeSteps	[*int:int*]	new timestep range context
Region	[*int*]	new region context
Regions	[*int:int*]	new region range context
Fishery	[*fisheryname*]	new fishery context
Fishery	[*fisheryabbrev*]	new fishery context
Fishery	[*int*]	new fishery context
Fisheries	[*int:int*]	new fishery range context
Ceiling	[*forceflag*] [*float*]	these fields specify the force flag and quota, respectively Valid values for *forceflag* are "forced" and "unforced."

Note that the "Ceiling" token is the only simple token valid in this context. Since this context is a variation on a generic array command block, the "end" token for any innermost block will be subsumed by the "Ceiling" token, just as in any other generic array.

3.9 CNRData Tokens

The token "CNRData" is a generic array token that enters a new context for providing Chinook non-retention (CNR) data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
CNR_Method	[*cnr method specifier*] [*command block data*] end	new CNR method context

Since there are no simple tokens associated with this generic array (only the CNR_Method command block token), "end" tokens are required for all levels of the generic array.

3.9.1 CNR Methods

The "CNR_Method" token enters a new context depending on the accompanying *cnr method specifier*. Each CNR_Method command block causes the creation of a CNR object of the corresponding type, which will be assigned to all applicable locations in the generic array. Valid *cnr method specifiers* and the subtokens valid within each context are:

cnr method specifier: HarvestRatio

Token	Data	Meaning
CNRLegalSelectivity	[*float*]	legal selectivity value
CNRSubLegalSelectivity	[*float*]	sublegal selectivity value

cnr method specifier: SeasonLength

Token	Data	Meaning
CNRLegalSelectivity	[*float*]	legal selectivity value
CNRSubLegalSelectivity	[*float*]	sublegal selectivity value

cnr method specifier: ReportedEncounter

Token	Data	Meaning
CNRLegalSelectivity	[*float*]	legal selectivity value
CNRSubLegalSelectivity	[*float*]	sublegal selectivity value
LegalEncounters	[*float*]	legal encounters
SubLegalEncounters	[*float*]	sublegal encounters
LandedCatch	[*float*]	landed catch

cnr method specifier: HarvestRatioMultipleEncounter

Token	Data	Meaning
CNRLegalSelectivity	[*float*]	legal selectivity value
CNRSubLegalSelectivity	[*float*]	sublegal selectivity value
BaseSeason	[*float*]	base season length
RecaptureInterval	[*float*]	recapture interval
LegalReleaseMortRate	[*float*]	legal release mortality rate

cnr method specifier: SeasonLengthMultipleEncounter

Token	Data	Meaning
CNRLegalSelectivity	[*float*]	legal selectivity value
CNRSubLegalSelectivity	[*float*]	sublegal selectivity value
BaseSeason	[*float*]	base season length
RecaptureInterval	[*float*]	recapture interval
LegalReleaseMortRate	[*float*]	legal release mortality rate
LegalSeason	[*float*]	legal season length
CNRSeason	[*float*]	cnr season length

Example:

```
CNRData FisheryXyear

Fishery 1 # Alaska T
  Year 1981
    CNR_Method                ReportedEncounter
      CNRLegalSelectivity     0.34
      CNRSubLegalSelectivity  1
      LegalEncounters         18225
      SubLegalEncounters      18578
      LandedCatch             248791
    end CNR_Method
```

```
  end Year
end Fishery
end CNRData
```

3.10 Cohorts

The “Cohorts” token enters a new context for the specification of initial model cohorts. There is one token valid at this level:

Token	Data	Meaning
Stock	[*stock name*] [*command block data*] end	new context for specifying initial cohorts of the specified stock

Within the “Stock” context there is one valid token:

Token	Data	Meaning
Cohort	[*int*] [*float*] [*int*] [*command block data*] end	brood year, initial abundance, initial region; new context for specifying characteristics of this initial cohort

Within the “Cohort” context the following tokens are valid. Note that these simple tokens are unusual in that they have no data fields. They are all optional, and the cohort will assume the given default values if not specified. In cases where more that one token share the same meaning, they are mutually exclusive, and the token encountered last will be the value assigned to the cohort.

Token	Data	Default	Meaning
Wild	[*none*]	*stock value*	production type
Hatchery	[*none*]	*stock value*	production type
Immature	[*none*]	Immature	maturation status
Mature	[*none*]	Immature	maturation status
Marked	[*none*]	Unknown	mark status
Unmarked	[*none*]	Unknown	mark status
Tagged	[*none*]	Unknown	tag status
Untagged	[*none*]	Unknown	tag status
Male	[*none*]	Unknown	sex
Female	[*none*]	Unknown	sex
G1	[*none*]	Unknown	growth group
G2	[*none*]	Unknown	growth group
G3	[*none*]	Unknown	growth group
G4	[*none*]	Unknown	growth group
G5	[*none*]	Unknown	growth group

Example:

```
Cohorts
  Stock "Alaska South SE"  # Initial cohorts for stock number: 1
    Cohort     1973  3345.02050867604  1
    end Cohort
  end Stock
end Cohorts
```

3.11 FPData Tokens

The token “FPData” is a generic array token which enters a new context for providing “fp” (fishery policy) data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
FP	[*float*]	fp value

Example:

```
FPData FisheryXyearXstockXage

Fishery 1   # Alaska T

  Year 1979        # For fishery: Alaska T
    Stock AKS #  Alaska South SE
      Age 3 FP 1
      Age 4 FP 1
      Age 5 FP 1
      Age 6 FP 1
    end Stock
  end Year
end Fishery
end FPData
```

3.12 FisherySchedule Tokens

The “FisherySchedule” token begins a new context used for specifying information as to when fisheries are active. It functions as a *generic array token*, except that no *generic array dimension specifier* is used, and valid tokens are restricted to only certain *generic array subtokens*. Valid tokens in this context are:

Token	Data	Meaning
Year	[*int*]	new year context
Years	[*int:int*]	new year range context
TimeStep	[*int*]	new timestep context
TimeSteps	[*int:int*]	new timestep range context
Region	[*int*]	new region context
Regions	[*int:int*]	new region range context
Fishery	[*fisheryname*]	new fishery context
Fishery	[*fisheryabbrev*]	new fishery context
Fishery	[*int*]	new fishery context
Fisheries	[*int:int*]	new fishery range context
Active	[*none*]	fishery is active in this time and region

Note that the “Active” token is the only simple token valid in this context. Since this context is a variation on a generic array command block, the “end” token for any innermost block will be subsumed by the “Active” token, just as in any other generic array.

Example:

```
FisherySchedule

  Fishery 1       # Alaska T
    TimeStep 1     # PreTerminal timestep
      Region 1     # PreTerminal region
        Active
    end TimeStep
  end Fishery
end FisherySchedule
```

3.13 MaturationData Tokens

The token "MaturationData" is a generic array token which enters a new context for providing maturation rate data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
MatRate	[*float*]	maturation rate

Example:

```
MaturationData StockXyearXageXtimeXregion

# No maturation occurs during timesteps 2:4.
TimeSteps 2:4 MatRate 0.0

# Maturation only occurs at the end of the first timestep.
TimeStep 1
  Regions 2:4 MatRate 0.0
  Region 1  # Preterminal region.
    Age 1 MatRate 0.0

    # Data for stocks with fixed maturation rates for all years.
    # Years 1979:1999

    Stock NTH # All years; run type = Spring
      Age 2 MatRate 0.0
      Age 3 MatRate 0.051396523
      Age 4 MatRate 0.14471728
      Age 5 MatRate 0.69004977
      Age 6 MatRate 0.99999988
    end Stock
  end Region 1
end TimeStep 1
end MaturationData
```

3.14 NatMortRateData Tokens

The token "NatMortRateData" is a generic array token which enters a new context for providing natural mortality rate data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
NaturalMortality	[*float*]	natural mortality rate

Example:

```
NatMortRateData StockXyearXageXtimeXregion

TimeStep 1
  Region 1
    Stock 1 # Alaska South SE // Run type = Spring
      Age 1 NaturalMortality 0.0
      Age 2 NaturalMortality 0.5
      Age 3 NaturalMortality 0.4
      Age 4 NaturalMortality 0.3
      Age 5 NaturalMortality 0.2
      Age 6 NaturalMortality 0.1
    end Stock
  end Region 1
end Timestep1
end NatMortRateData
```

3.15 PnvData Tokens

The token "PnvData" is a generic array token which enters a new context for percent non-vulnerable data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
PNV	[*float*]	percent non-vulnerable

Example:

```
PnvData FisheryXyearXrunXage

# Data for fisheries that have constant PNVs for all years.

Fishery 2   # Variable PNV fishery: North T
  Years 1979:1986
    Run Fall
      Age 2 PNV              0.5938
      Age 3 PNV              0.3868
      Age 4 PNV              0.0332
      Age 5 PNV              0.0049
    end Run
    Run Spring
      Age 3 PNV              0.5938
      Age 4 PNV              0.3868
      Age 5 PNV              0.0332
      Age 6 PNV              0.0049
```

```
    end Run
  end Years
end Fishery
end PnvData
```

3.16 ProductionFunctions Tokens

The token "ProductionFunctions" is a generic array token which enters a new context for providing production function data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
Production	[*production function type*] [*production function data*]	production function selection production data

The nature of the *production function data* varies depending on the particular production function specified. The "Production" token is a special type of token that does not subsume the "end" token for the enclosing level of the generic array. Thus, "end" tokens are required for all levels of the generic array. Multiple "Production" functions may be specified simultaneously at a given level.

3.16.1 Production Function Types

The following are the valid *production function types* and the associated *production function data* for each. Please note that for a given production function, all of the specified data fields are required and must appear in the proper order.

Production Function Type Token	Data	Meaning
Linear	[*float*]	minimum number of spawners
	[*float*]	maximum number of spawners
	[*float*]	ev value
	[*float*]	slope
Ricker	[*float*]	minimum number of spawners
	[*float*]	maximum number of spawners
	[*float*]	ev value
	[*float*]	Ricker A value
	[*float*]	Ricker B value
	[*float*]	recruits to age 1 ratio
EnhancedRicker	[*float*]	minimum number of spawners
	[*float*]	maximum number of spawners
	[*float*]	ev value
	[*float*]	Ricker A value
	[*float*]	Ricker B value
	[*float*]	recruits to age 1 ratio
	[*int*]	density dependence flag (0=false;1=true)
	[*float*]	productivity parameter
	[*float*]	smolt at age 1

Production Function Type Token	Data	Meaning
EnhancedRicker (continued)	[*float*]	max brood proportion
	[*float*]	smolt production change
VariableTruncationRicker	[*float*]	minimum number of spawners
	[*float*]	maximum number of spawners
	[*float*]	ev value
	[*float*]	Ricker A value
	[*float*]	Ricker B value
	[*float*]	recruits to age 1 ratio

Example:

```
ProductionFunctions StockXyearXtimeXregion  #Production functions by
  #stock and year.
TimeStep 4
Region 4
StockNum 9 #Production functions for stock: GSH
  Year 1979        #For stock: GSH
    Production Linear  0  5318  0.576309919  101.088866871763
    Production Ricker  5318  10318  0.576309919  2.813
72371.2428651556  0.181012304888616
  end Year
  Year 1980        #For stock: GSH
    Production Linear  0  5318  0.338252485  101.088866871763
    Production Ricker  5318  10318  0.338252485  2.813
72371.2428651556  0.181012304888616
  end Year
  Year 1981        #For stock: GSH
    Production Linear  0  5318  1.36516976  101.088866871763
    Production Ricker  5318  10318  1.36516976  2.813
72371.2428651556  0.181012304888616
  end Year
  Year 1982        #For stock: GSH
    Production Linear  0  5054.47415595914  1.3196733  101.088866871763
  end Year
  Year 1983        #For stock: GSH
    Production Linear  0  5318  0.560896635  101.088866871763
    Production Linear  5318  5562.0650328853  0.560896635
17.5666110352434
    Production Ricker  5562.0650328853  10562.0650328853  0.560896635
2.813  72371.2428651556  0.181012304888616
  end Year
end Stock
end Region
end TimeStep
end Production
```

3.17 ShakerData Tokens

The token "ShakerData" is a generic array token which enters a new context for shaker mortality data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
Method	[*shaker method specifier*] [*command block data*] end	new shaker method context

Since there are no simple tokens associated with this generic array (only the Method command block token), "end" tokens are required for all levels of the generic array.

3.17.1 Shaker Methods

The "Method" token enters a new context depending on the accompanying *shaker method specifier*. Each Method command block causes the creation of a shaker object of the corresponding type, which will be assigned to all applicable locations in the generic array. Valid *shaker method specifiers* and the subtokens valid within each context are:

shaker method specifier: Simple

Token	Data	Meaning
SubLegalReleaseMortRate	[*float*]	sublegal release mortality rate

cnr method specifier: SimpleDrop

Token	Data	Meaning
SubLegalReleaseMortRate	[*float*]	sublegal release mortality rate
DropOffRate	[*float*]	drop-off mortality rate

shaker method specifier: Custom

Token	Data	Meaning
SubLegalReleaseMortRate	[*float*]	sublegal release mortality rate
VulnerabilityTable	[*generic array dim spec*] [*data*] end	vulnerability table data

cnr method specifier: CustomDrop

Token	Data	Meaning
SubLegalReleaseMortRate	[*float*]	sublegal release mortality rate
DropOffRate	[*float*]	drop-off mortality rate
VulnerabilityTable	[*generic array dim spec*] [*data*] end	vulnerability table data

3.17.2 VulnerabilityTable Tokens

The token "VulnerabilityTable" is a generic array token which enters a new context for entering shaker vulnerability data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
Vulnerable	[*none*]	cohort is vulnerable to the given fishery
Invulnerable	[*none*]	cohort is not vulnerable to the given fishery

Example:

```
ShakerData FisheryXyear

Fishery 6    # Geo St T
  Method  SimpleDrop
   SubLegalReleaseMortRate  0.255
    DropOffRate              0.017
  end Method
end Fishery

Fishery 7    # Alaska N
  Method  CustomDrop
    SubLegalReleaseMortRate  0.9
    DropOffRate              0
    VulnerabilityTable StockXageXtime
      TimeStep 1  # Preterminal
  Ages 3:4   # Spring stocks.
        Stock 1 Vulnerable # Alaska South SE
        Stock 2 Vulnerable # North/Centr
        Stock 14 Vulnerable # Nooksack Spring
        Stock 24 Vulnerable # Willamette R
        Stock 25 Vulnerable # Spr Cowlitz Hat
      end Ages
    end TimeStep
    end VulnerabilityTable
  end Method
end Fishery
end ShakerData
```

3.18 TransitionMatrix Tokens

The token "TransitionMatrix" is a generic array token which enters a new context for entering transition matrix data. In addition to all *generic array subtokens,* the following tokens are valid in this context:

Token	Data	Meaning
Data	[*transition matrix data*]	cohort transition matrix

3.18.1 Transition Matrix Data

The transition matrix data is a sequence of *n* X *n* [*float*] entries, where *n* is the number of regions specified in the "Configuration" context. Altogether the matrix is taken as a simple token, and thus subsumes the "end" for the innermost generic array level.

Example:

```
TransitionMatrix RunXageXmaturityXtime

Run Spring
  Immature  # Immature cohorts.
    Age 0
        Data
          1.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  1.0
    Age 1
        Data
          1.0  0.0  0.0  1.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  0.0
    Ages 2:6
        Data
          1.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  1.0
  end Maturity
end Run
end TransitionMatrixOutput Language
```

4 Output Language

4.1 Overview

Pacific salmon researchers and managers use many computer programs and models. There is a need to communicate information between programs and from individual programs to users. Three general data communication methods are:

1. Raw data from one program is used by other programs for further analysis (e.g., FRAM to TAMM communication);
2. Raw data from one program is used by auxiliary programs to create formatted reports (e.g., PSC Chinook Model raw data dump); or
3. Each program creates model specific formatted reports (e.g., PSC Chinook Model and FRAM formatted reports).

PSC Chinook Model users have found Method 2 to be effective in creating reports and analyzing output data. As the program moves through a simulation, raw data are output to an ASCII text file in a specific format. Any auxiliary programs must know the format of the output file in order to use the data. The intent of the Coast Model is to adopt a standardized output format to improve data communication within and between programs.

The basic idea is to stream standardized data sentences from a model to a text file. Then anyone who knows the formats can write a program using any type of software and hardware to read the data stream, extract the data of interest, and use that data as desired (e.g., further analysis or formatted report). The advantage of using standardized data sentences is that users can export any data in any order and at any interval. The disadvantage of this system is that there may be considerable duplication of information in each sentence, thus making for a large output file.

The following sections define the standardized data sentences used by the Coast Model. Additional sentences can be defined as necessary. We envision that one agency (e.g., NMFS or the Pacific States Marine Fisheries Commission) will supervise and maintain a list of standardized data sentences.

4.2 CohortID

Each cohort has a 14 digit unique identifier code composed of the parent stock abbreviation, the brood year (using all four digits for Y2K compatibility), and a string of single character codes representing each remaining cohort property in a fixed order (see table below). For example, the cohort identifier code URB1985FWIXXXX is a cohort from the URB stock, brood year 1985, wild production type, immature, and unknown mark, tag, sex, and growth group status.

Digit(s)	Permitted Values	Meaning
1-3	Stock Abbreviations	Stock identifier
4-7	Any	4 digit year
8	S,F	Spring (S) or Fall (F) stock
9	W,H	Wild (W) or Hatchery (H) stock
10	I,M	Immature (I) or Mature (M)
11	M,U,X	Mark status: Marked (M), Unmarked (U), Unknown (X)
12	T,U,X	Tag status: Tagged (T), Untagged (T), Unknown (X)
13	M,F,X	Sex: Male (M), Female (F), Unknown (X)
14	1,2,3,4,5,X	Growth Group: 1-5 or Unknown (X)

4.3 Output Sentences Supported by Coast Model

4.3.1 CABN sentence for cohort abundance

Field	Data	Sample
1	Sentence ID = CABN	CABN
2	Year	1985
3	TimeStep	3 (or TimeStep abbreviation)
4	Region	4 (or Region abbreviation)
5	CohortID	URB1986FWIXXXX
6	StartAbn	378947.1438 (a double)

Example:

```
CABN, 1979, 1, 1, STL1974FWIXXXX, 105.468
CABN, 1979, 1, 1, STL1975FWIXXXX, 1616.72
CABN, 1979, 1, 1, STL1976FWIXXXX, 4267.45
CABN, 1979, 1, 1, STL1977FWIXXXX, 6578.15
CABN, 1979, 1, 1, STL1978FWIXXXX, 13283.2
CABN, 1979, 1, 1, VGT1976FHIXXXX, 6815.32
CABN, 1979, 1, 1, VGT1977FHIXXXX, 72867.8
CABN, 1979, 1, 1, VGT1978FHIXXXX, 164354
CABN, 1979, 2, 2, STL1974FWIXXXX, 31.6403
CABN, 1979, 2, 2, STL1975FWIXXXX, 485.017
CABN, 1979, 2, 2, STL1976FWIXXXX, 1280.24
CABN, 1979, 2, 2, STL1977FWIXXXX, 1973.44
CABN, 1979, 2, 2, STL1978FWIXXXX, 3984.95
CABN, 1979, 2, 2, VGT1976FHIXXXX, 681.532
```

4.3.2 NMRT sentence for natural mortality

Field	Data	Sample
1	Sentence ID = NMRT	NMRT
2	Year	1985
3	TimeStep	3 (or TimeStep abbreviation)
4	Region	4 (or region abbreviation)
5	CohortID	URB1986FWIXXXX
6	Mortality	25.283949 (a double)

Example:

```
NMRT, 1979, 2, 2, STL1974FWIXXXX, 0.2766
NMRT, 1979, 2, 2, STL1975FWIXXXX, 8.93546
NMRT, 1979, 2, 2, STL1976FWIXXXX, 37.493
NMRT, 1979, 2, 2, STL1977FWIXXXX, 82.2433
NMRT, 1979, 2, 2, STL1978FWIXXXX, 223.659
NMRT, 1979, 2, 2, VGT1976FHIXXXX, 19.9594
NMRT, 1979, 2, 2, VGT1977FHIXXXX, 303.677
NMRT, 1979, 2, 2, VGT1978FHIXXXX, 922.452
NMRT, 1979, 2, 3, STL1974FWIXXXX, 0.3688
NMRT, 1979, 2, 3, STL1975FWIXXXX, 11.914
```

4.3.3 FMRT sentence for fishing mortality

Field	Data	Sample
1	Sentence ID	FMRT
2	Year	1985
3	TimeStep	3 (or TimeStep abbreviation)
4	Region	4 (or region abbreviation)
5	Fishery	2 (or fishery abbreviation)
6	CohortID	URB1986FWIXXXX
7	LegalCatch	345.192837 (a double)
8	ShakerMort	43.1288393 (a double)
9	DropOffMort	5.68459 (a double)
10	CNRSubLegalMort	15.7398 (a double)
11	CNRMort	21.1093994 (a double)
12	EffortScalar	0.95 (a double)

Example:

```
FMRT, 1983, 7, 5, 13, STL1978FWMXXXX, 1.33668, 0, 0, 0, 0, 1
FMRT, 1983, 7, 5, 13, STL1979FWMXXXX, 17.7754, 0, 0, 0, 0, 1
FMRT, 1983, 7, 5, 13, STL1980FWMXXXX, 3.80355, 0, 0, 0, 0, 1
FMRT, 1983, 7, 5, 13, STL1981FWMXXXX, 0.0863695, 0, 0, 0, 0, 1
FMRT, 1983, 7, 6, 13, STL1978FWMXXXX, 0.765205, 0, 0, 0, 0, 1
FMRT, 1983, 7, 6, 13, STL1979FWMXXXX, 10.8742, 0, 0, 0, 0, 1
FMRT, 1983, 7, 6, 13, STL1980FWMXXXX, 2.22352, 0, 0, 0, 0, 1
FMRT, 1983, 7, 6, 13, STL1981FWMXXXX, 0.0494872, 0, 0, 0, 0, 1
FMRT, 1983, 7, 7, 13, STL1978FWMXXXX, 0.222199, 0, 0, 0, 0, 1
FMRT, 1983, 7, 7, 13, STL1979FWMXXXX, 3.65069, 0, 0, 0, 0, 1
```

4.4 Proposed Output Sentences For Future Use

4.4.1 CMIG sentence for cohort migration

Field	Data	Sample
1	Sentence ID = CMIG	CMIG
2	Year	1985
3	TimeStep	3 (or TimeStep abbreviation)
4	FromRegion	4 (or region abbreviation)
5	ToRegion	6 (or region abbreviation)
6	NumMigrants	25.49304958 (a double)

4.4.2 SABN sentence for stock abundance in a region

Field	Data	Sample
1	Sentence ID	SABN
2	Year	1985
3	TimeStep	3 (or TimeStep abbreviation)
4	Region	4 (or region abbreviation)
5	StockID	URB
6	StartAbn	378947.1438 (a double)

4.4.3 CLHD sentence for cohort life history data

Field	Data	Sample
1	Sentence ID = CLHD	CLHD
2	Year	1985
3	TimeStep	3 (or TimeStep abbreviation)
4	Region	4 (or region abbreviation)
5	CohortID	URB1986FWIXXXX
6	StartAbn	28394.1928 (a double)
7	NatMort	28.3941928 (a double)
8	TotCat	3564.19283 (a double)
9	TotShakers	32.9238287 (a double)
10	TotCNR	27.3984749 (a double)
11	TotOutMig	283.193993 (a double)
12	TotInMig	11.9384773 (a double)
13	EndAbn	23392.1958 (a double)

4.5 Generating the Output Data File

At the end of each process during a timestep the DataRequestManager has the opportunity to stream data to a data file (named *coast_output.txt*). To save space, null data (e.g., zero catches, zero natural mortalities, zero migrations) are not be streamed. Currently, the type and frequency of data streaming to the data file is fixed (i.e., not configurable by the user). Future versions of the model should allow the user to configure the output data streaming. See Section 5.5.3.3 for further details.

5 Code Description

5.1 Introduction

This document provides a general overview of the primary modules, classes, and structures used in the Coast model. It is not intended to provide a detailed listing of all classes, variables, and routines, as such information can best be distilled by using any software browsing tool designed for that purpose. Class-level documentation is provided as inline comments, usually in the .h file declaring the class.

It is assumed that the reader of this document is familiar with the overall function and approach of the simulation engine, as described in Section 2 Coast Model Processes. This section focuses on issues relevant to the design and implementation of the code itself.

5.2 Naming Conventions

In accordance with common C++ practice, class names typically consist of sequences of words run together and capitalized, e.g. **FisheryQuotaPolicy**. Class declarations appear in .h files corresponding to the class name, and definitions are in .cpp files of the same name. Exceptions occasionally occur for classes that operate very closely together, in which case more than one related class declaration or definition may appear in the same file.

Class methods are named with sequences of words run together with all except the first word capitalized, as in **makeHarvestList()**. Class member variables typically are also sequences of words with all but the first word capitalized, and with an underscore appended, as in **legalReleaseMortRate_**.

Global variables and routines with external linkage are capitalized.

There are exceptions to these rules, particularly in some of the older sections of code written before these particular naming conventions became common in the C++ community.

5.3 Class Overview

Many classes and objects participate in the Coast Model application. The more important and pervasive ones are listed here and described in more detail in subsequent sections.

5.3.1 Monostates and Managers

A monostate is a pattern that describes a class where all methods and members are static. It is a variation on the Singleton pattern[1] and is used to provide the effect of a globally accessible object. The predominant paradigm of input parameter organization in the Coast model is to have a number of monostates that control the main Simulation Processes and house most of the parameter data in tables. For clarity of usage, methods and members of monostates are typically accessed directly through the use of the scope operator ('::'), without creating a local object. The most common form of monostate in the Coast model is the "manager", of which there are a number, correspondingly roughly to the simulation processes. The following is a list of the managers in the Coast model: **IterationManager, CohortManager, DataRequestManager, NaturalMortalityManager, FisheryManager, HarvestManager, PolicyControlManager, MaturationManager, SpawningManager, MigrationManager**, and **StandardDataOutputManager.**

---

[1] Gamma, Helm, Johnson, Vlissides, "Design Patterns", pp. 127-134

Other monostates providing important functions are **Geographer** and **Historian**.

These classes are described in more detail elsewhere in this document.

5.3.2 Globals

Three classes are currently implemented as global objects. **Fisheries** and **Stocks** are anachronistic, and it is expected that eventually these objects would be moved into the **FisheryManager** and **StockManager** (not yet extant), respectively. **SystemClock** is a global object of type **Chronograph** that keeps track of time during the simulation**.**

5.3.3 Other Important Classes

Other classes that carry out fundamental tasks in the model include: **Parser, GenericArray, GenericArrayIndex, Cohort, CohortID, Fishery, Stock, DataRequest, IterationControl,** and **State.** Additionally, each of the Simulation Processes utilizes a number of classes to perform its task. The harvest process is particularly complex and warrants special attention. Important classes in that module include: **FisheryUnit, FisheryPolicy, HvMort,** and **HarvestProcess.** These classes are examined in more detail below.

5.4 Process Overview

The entry points for the model code are found in file *c2main.cpp*. These consist of **main()** for a Windows console application, or **activeXentry()** for an ActiveX component. In very brief (and with some details elided), the process level overview of a single simulation run is as shown below. Filenames appear in braces.

main()	{ *c2main.cpp* }
ModelConfig()	{ *init.cpp* }
RunTheModel()	{ *engine.cpp* }
SimulationWrapup()	{ *init.cpp* }
ModelCleanup()	{ *init.cpp* }

Other sections of this document will examine each of the above core processes in more detail.

5.4.1 Simulation Processes

As described in Section 2, the simulation engine consists of a nested looping of years and timesteps, within which each of the Simulation Processes is executed in turn. These Simulation Processes presently consist of the following calls (from "engine.cpp"):

```
CohortManager::ageCohorts();
    NaturalMortalityManager::takeNaturalMortality();
    FisheryManager::takeHarvests();
    MaturationManager::maturateCohorts();
    SpawningManager::spawnCohorts();
    MigrationManager::migrateCohorts();
```

5.5 Class and Object Detail

This section provides further information on some of the major classes and objects used in the Coast Model.

5.5.1 Managers and Other Globals

The following table describes the basic duties of each of the Managers.

Table 3 Managers and other global classes and objects

CohortManager	Creation and ageing of cohorts.
NaturalMortalityManager	Cohort natural mortality processes and data.
FisheryManager	Entry point for harvest process and storage of FisheryPolicy data.
HarvestManager	Harvest rate data storage and calculation of harvest sub-processes (shak CNR, etc.).
MaturationManager	Cohorts maturation processes and data.
PolicyControlManager	Manages PolicyControl objects used for some types of harvest management.
SpawningManager	Cohort spawning processes and data.
MigrationManager	Cohort migration processes and data.
Fisheries	Fishery object storage.
Stocks	Stock object storage.
IterationManager	Handles iteration across timesteps.
DataRequestManager	Summary and output data calculation and storage.
StandardDataOutputManager	Outputs Standard Data Sentences to file.
SystemClock	Manages system time (timesteps and years).
Geographer	Manages information about geographic regions.
Historian	Makes certain common summary data available to all processes.

5.5.2 Common Fundamental Objects

The following sections describe some of the common objects used throughout the model.

5.5.2.1 Cohort and CohortID

Cohort is one of the most basic classes in the model. One object of this class is created for every cohort in the system. The purpose of this object is twofold: to store identification information about a particular cohort in the simulation, and to store and manage information about that cohort's regional abundance as the simulation progresses.

Cohort identification information is handled by the **CohortID** class (because of its close association with **Cohort**, it is also declared in *Cohort.h.*) Every **Cohort** object contains a constant **CohortID**. This very simple class contains all the identifying information concerning a cohort, with each characteristic typically being implemented as an instance of an enumerated type. The **Cohort** class duplicates the interface of the **CohortID** class and thus serves as an "envelope"[2] for it. Existing cohort characteristics include **stock_, broodYear_, age_, run_**, **prodType_, maturity_, mark_, tag_, sex_,** and **growthGroup_**. Of these, **run_** refers to the run timing (spring or fall) and **prodType_** refers to the production type (wild or hatchery). Otherwise, the meanings of these identifiers should be self-evident (refer to the code for more details). See below for an example of how to add a new type of cohort characteristic to the model.

The Cohort class maintains the regional abundances for a cohort in a simple contained vector of type **AbundanceVector**. For active cohorts, this represents the current abundance distribution for that cohort across all the regions at any given moment during the simulation. Methods for accessing and manipulating these abundances are also provided. The temporal granularity for abundance calculations is the "Simulation Process" (Section 5.4.1), meaning that it is required that all cohort abundances be current and correct before entering and after exiting each Simulation Process. Note that since cohort abundances are accurate at this moment between Simulation Processes, but will be updated during the subsequent Simulation Process, the **DataRequestManager** is called at each of these times to record (if configured to do so) cohort abundances for the purposes of data output.

The **Cohort** class also contains an **AbundanceVector** for the purpose of storing mortalities calculated during the natural mortality Simulation Process. This is an anachronistic implementation maintained for convenience. In the long run this data would be better handled by new structures in the **NaturalMortalityManager**.

Cohort objects are created and managed by the **CohortManager**, which serves (among other things) as a factory (i.e. implements a modified Factory Pattern[3]) for **Cohort** objects. The **CohortManager** also provides access to all **Cohort** objects. At configuration time, the initial set of **Cohort** objects is created according to specifications given by the user in the input data files. These cohorts maintained by the **CohortManager** on a pending list. At the start of each timestep, the **CohortManager** determines which pending cohorts should be activated based on the brood year of the cohort, and activates the cohort if appropriate. New cohorts created during the course of the simulation (due to maturation or spawning, for example) are typically activated immediately. The **CohortManager** also conceptually maintains the list of active cohorts. However, the current implementation has the contents of this list distributed over the **Stock** objects. All handles to cohorts should be acquired through the **CohortManager**, although in older sections of code there are still a small number of anachronistic references to active cohorts obtained directly from the **Stock** objects.

5.5.2.2 GenericArray and GenericArrayIndex

The Generic Array system used in the Coast Model is fundamental to the storage of input parameter data. This system is employed to achieve maximum flexibility in the specification of parameter dimensionality at runtime, while still providing rapid data access during the simulation. The basic concept is that there are a fixed number of possible dimensions for parameters, and that indices for all known dimensions are provided during lookup.

---

[2] Coplien, "Advanced C++", pp. 133-140

[3] Gamma, Helm, Johnson, Vlissides, "Design Patterns", pp. 107-116

However, the dimensionality for a given parameter is fixed at initialization time (based on input file data), and the parameter tables configure themselves appropriately, making use only of the correct indices when accessing data elements. The **GenericArray** abstract base class provides the interface for data parameter tables exhibiting this functionality, and the **GenericArrayIndex** concrete class encapsulates the indices for all known dimensions for the purpose of indexing a particular element in the array.

Possible dimensions dereferenced by a **GenericArray** are:

- Cohort characteristics: stock, brood year, run type (fall or spring), production type (wild or hatchery), maturity, mark status, tag status, sex, growth group, age;
- Process parameters: fishery, region;
- Clock parameters: year, timestep.

All possible dimensions are contained in the **GenericArrayIndex**, which provides a number of constructors. The most basic **GenericArrayIndex** constructor is (from *GenericArray.h*):

```
GenericArrayIndex(const Cohort* c, unsigned fishery=0,
       unsigned region=0, unsigned year=0, unsigned time=0);
```

All of the cohort related indices are initialized from the cohort object, while indices into the global lists of fisheries and regions may also be supplied, as well as year and timestep references.

Since the purpose of the Generic Array system is to allow parameter tables to change dimensionality at runtime, it is important to always supply information about every possible dimension during table lookup. Following is an example table lookup from the **HarvestManager**.

Example:

```
// base harvest rate * fp
double HarvestManager::
hvRate(const HvMort* hvmort)
{
    GenericArrayIndex id(&(hvmort->cohort()),
                           hvmort->fishery().index(),
                           hvmort->region().index(),
                           ::SystemClock->currentYearIndex(),
                           ::SystemClock->currentTimeIndex());

    return ((*hvRateTable_)[id] * (*fpTable_)[id]);
}
```

In this example, **hvRateTable_** and **fpTable_** are **GenericArray** objects representing tables of base harvest rates and FP values respectively. In the last line, each is dereferenced (using **operator[]**) with the local **GenericArrayIndex** object **id**. The constructor shown above is used to create **id**, utilizing information from the passed in **HvMort** pointer to obtain the proper cohort, fishery, and region. Note that **Fishery** and **Region** objects are each able to supply the proper index for use in Generic Array lookups. The global **SystemClock** is used to obtain the current simulation year and timestep. Thus, regardless of what the actual dimensionalities of the harvest rate and FP tables are (as defined by the user in configuration files) the proper data will be extracted, because indices for all possible dimensions are supplied.

Classes that derive from GenericArray are straightforward. **FisheryYearArray** is an example of a two dimensional generic array that uses the fishery and year dimensions for indexing. The definitions of the inherited lookup methods and private member data are (from *FisheryYearArray.h*):

```
template <class T>
class FisheryYearArray : public GenericArray<T> {
      //…
      // generic interface
      virtual const T& operator[](const GenericArrayIndex& id) const
      { return array_[id.fisheryIndex][id.yearIndex]; }

      virtual T& operator[](const GenericArrayIndex& id)
      { return array_[id.fisheryIndex][id.yearIndex]; }
      //…

  private:
      size_t nFisheries_;
      size_t nYears_;

      Array2D<T> array_;
};
```

In this example, **array_** is a member variable of type **Array2D<T>** which provides a simple two dimensional array with the expected **operator[]** interface. The **FisheryYearArray** class also provides constructors(not shown above) to create the underlying array in the proper size and possibly initialize it with default values, as well as the required definition for the inherited abstract virtual method **newGenericIndexList()**. This last method is required in every concrete **GenericArray** derived class. Its purpose is to create one **GenericArrayIndex** on the heap for every possible element in the array and add each to the supplied list. This can be used by the caller to iterate over the entire array. For **FisheryYearArray** the definition of this method is:

```
// append to a list of GenericArrayIndex one new entry (on the heap) for
each index in the array.
template <class T>
void FisheryYearArray<T>::
newGenericIndexList(RWTPtrOrderedVector<GenericArrayIndex>& indexList) const
{
    for (int f=0; f<nFisheries_; ++f) {
        for (int y=0; y<nYears_; ++y) {
            GenericArrayIndex* id = new GenericArrayIndex(0, f, 0, y, 0);
            indexList.append(id);
        }
    }
}
```

In this example, **nFisheries_** and **nYears_** are class member variables initialized in the constructor to values corresponding to the proper sizes of the two dimensions.

5.5.2.3 GenericArrayFactory

Generic Arrays are created at runtime upon user request (from specifications in data input files) by using the **GenericArrayFactory** class. The class contains an enumerated type listing all **GenericArray** derived concrete classes currently available. More Generic Arrays may be added to this list as they are developed without disrupting the operation of existing ones. At present this enumeration consists of:

```
// all types of generic arrays
    enum GenericArrayType
    { TimeStep,
      StockAgeTime,
      StockYear, StockYearTimeRegion,
      StockYearAgeTime, StockYearAgeTimeRegion, StockYearAgeTimeMaturity,
      FisheryRegionTimeStockAge, FisheryRegionTimeStockAgeMark,
      FisheryYearAge, FisheryYearStock, FisheryYearStockAge,
         FisheryYear, FisheryYearRunAge, Fishery,
      RunAgeMaturityTime,
      MaturityTime,
      NTypes
    };
```

Note that the final value, **NTypes** is not an array, but rather an indicator of the end of the list. New Generic Arrays are obtained by calling one of the four factory methods made available by this class, shown below. These methods differ only in whether maximum dimension sizes are explicitly specified (by using a **GenericArrayIndex** parameter) and whether a default value for the elements of the array is specified. Function overloading was not used in the naming of these methods simply because at the time of coding the Microsoft compiler could not handle this language construct for template classes.

```
// build a new array of a particular dynamic type.  dimension
// sizes maxima by default, or are passed in the appropriate
// fields of the GenericArrayIndex parameter.  the caller is
// responsible for deleting this array.
GenericArray<T>*
newGenericArrayFn0(GenericArrayFactory<T>::GenericArrayType);

GenericArray<T>*
newGenericArrayFn1(GenericArrayFactory<T>::GenericArrayType,
                   const T& dfltVal);

GenericArray<T>*
newGenericArrayFn2(GenericArrayFactory<T>::GenericArrayType,
                   const GenericArrayIndex&);

GenericArray<T>*
newGenericArrayFn3(GenericArrayFactory<T>::GenericArrayType,
                   const GenericArrayIndex&, const T& dfltVal);
```

Whenever a new concrete **GenericArray** derived class is created which is to be made available to the user, the **GenericArrayFactory** class must be updated. The new array should be added to the **GenericArrayType** enumeration, and the constructor and the **newGenericArrayFn2()** and **newGenericArrayFn3()** methods updated.

The constructor for **GenericArrayFactory** creates the mapping between available array types and the actual strings that appear in user data files to request a particular type. Note also that in the constructor there is a special check that is activated in debug compilations to provide increased assurance that all is in order:

```
#ifdef DEBUG
    // make sure all the types are accounted for
    assert(MaturityTime == 16 && MaturityTime == NTypes-1);
#endif // DEBUG
```

This check must be also be updated. The convention is to always keep **MaturityTime** as the last array in the enumeration, and update its index accordingly here.

See the **HarvestParser** example below for an illustration of how to use **GenericArrayFactory**.

5.5.2.4 Example: Adding a New Cohort Characteristic

Although highly configurable during runtime, there are obviously some Coast model modifications which would need to occur at the code level. One example is adding a new cohort characteristic. This section shows all the steps necessary to implement a change of this sort.

Let's assume a new cohort characteristic, "color" is to be added. First, the characteristic is added to the **CohortID** class found in *Cohort.h,* as in:

```
enum Color { C_UNKNOWN, REDFISH, BLUEFISH, N_COLORS };
```

A new member variable, **color_** would be added to the private sections of **CohortID** and the initializer lists in all of the constructors in the **CohortID** class would be updated. The **operator==()** method and the **idString()** methods for this class would also be updated. A new accessor method would also be added:

```
Color color() const { return color_; }
```

Next, a similar accessor method would be added to the **Cohort** class:

```
CohortID::Color color() const { return id_.color_; }
```

Since Generic Arrays work closely with cohort characteristics, the **GenericArrayIndex** class in *GenericArray.h* must also be updated. A new member variable for indexing on color would be added:

```
unsigned color;
```

The initializer lists in all **GenericArrayIndex** constructors would be updated to reflect the new field. The method **operator==()** would also be updated. The member method **resetToSizes()** would be updated by adding the following new line:

```
color = CohortID::N_COLORS;
```

This completes the basic modifications necessary in order to use the new cohort characteristic in new algorithms. However, it is also necessary to make modifications to allow the user to specify this characteristic as part of the cohort definitions in the input files. To achieve this, new methods are added for each of the possible new characteristic values in the **CohortParser** class (see the Harvest Parser example in Section 5.5.3.4.1 for a more general explanation of the operation of the **Parser** and its derived classes):

```
void redfish();
void bluefish();
```

The definitions for these methods in *CohortParser.cpp* are straightforward:

```
void CohortParser::redfish()
```

```
{  cohortID_.color_ = CohortID::REDFISH; }
void CohortParser::bluefish()
{  cohortID_.color_ = CohortID::BLUEFISH; }
```

Additionally, new tokens to specify these values are declared in *CohortTokens.h* and defined in *CohortTokens.cpp*, and included in **CohortParser::initKeymap()**, e.g.:

```
keymap_.insert(CohortTokens::REDFISH, redfish);
keymap_.insert(CohortTokens::BLUEFISH, bluefish);
```

Finally, the user needs the ability to specify these characteristic values as possible dimensions in input files that handle Generic Arrays (on the assumption that eventually a concrete Generic Array will be built which offers this dimension). This is done by modifying the **GenArrayParser** class. New methods are added for each of the new cohort characteristic values:

```
void GenArrayParser::redfish()
void GenArrayParser::bluefish()
```

A new method to push state for the color dimension is also added:

```
void pushStateColor(const Range<int> colors);
```

The definitions for similar methods in *GenArrayParser.cpp* should be examined for examples on how to implement these methods. Member method

```
template<class T> void initGenArrayKeymap(KeywordProcessor<T>& keymap)
```

is modified by adding code to connect the new characteristic value methods with the proper tokens (this method is found in *GenArrayParser.h*):

```
keymap.insert(CohortTokens::REDFISH, &T::redfish);
keymap.insert(CohortTokens::BLUEFISH, &T::bluefish);
```

This completes the implementation of the new cohort characteristic.

5.5.2.5 Iteration

The Coast Model can handle iteration over timesteps (and years) during the course of simulation. This feature is typically used to achieve particularly complex harvest management objectives. In the Coast Model, an iterative algorithm begins operating in a fixed and predetermined first timestep, and completes in a fixed last timestep. The process of iteration is straightforward: when the first timestep is reached the state of the system is recorded. When the last timestep is reached, an analysis is carried out to determine whether or not iteration is required. If iteration is required, then the system clock is reset to the first timestep, iteration control variables are modified with new data, and the previously recorded state is restored. If iteration is not required, then the simulation proceeds in a normal manner. A number of classes work together to implement this type of simulation iteration, including **IterationManager**, **IterationControl**, and **State**. The reader should examine comments in the code for these classes and derived classes for detail concerning the working of this system beyond the brief overview provided below.

5.5.2.5.1 State

The concept of state is fundamental to the working of the iteration system. All data parameters in the system are divided into one of three categories:

<u>Static state:</u> data parameters whose value does not change over the lifetime of the simulation.

<u>Variable state:</u> non-control data parameters that may change value during the simulation.

<u>Control variables:</u> data parameters modified by iterative algorithms to achieve end conditions.

All **FisheryPolicy** objects, as kept in the **policyTable_** maintained by the **FisheryManager,** are considered control variables. These are the only control variables in the system, and thus are the only permissible data to be overwritten by **IterationControl** objects when making adjustments in order to meet end conditions. Most other input data parameters are part of the static state of the system. Variable state information includes all of the **Cohort** objects (which includes current abundances) and all of the **DataRequest** objects in the system (each of which may maintain local data.)

At the start of an iteration algorithm, the state of the system is saved. At some later point in time, if the end condition for that algorithm is not met, then the clock will be reset to the start time of the algorithm, the system state will be restored, and control variables will be modified so that the end condition might possibly be met on the upcoming iteration. The **State** class is the abstract class used for saving state information. The **SystemState** class records the state of the entire system for the purpose of later restoration as part of an iterative algorithm. Since static state information never changes, it is not saved as part of the **SystemState**. Since the control variables are modified in order to meet end conditions during iteration, and since multiple iterative algorithms may be operating simultaneously all modifying different subsets of the available control variables, none of the control variables are saved as part of the **SystemState**. Therefore, only variable state information is saved.

5.5.2.5.1.1 SystemState

The **SystemState** class is derived from the **State** abstract base class. As described above, it records all of the variable state information in the system. It employs the Composite[4] pattern in its implementation. Thus, it maintains a list of other **State** objects, which together encapsulate the entire variable state of the system. Since each **State** derived object maintains whatever local data is necessary and responds to messages to **saveState()** and **restoreState()** it is straightforward to carry out those operations for the entire system.

The **SystemState** class also implements an internal reference counting[5] system as a space optimization (see files *RCObject.h* and *RCPtr.h.*)

5.5.2.5.2 IterationControl

The basic abstract class involved in simulation iteration is **IterationControl**. It provides member data to track the first and last timestamps of the desired iterative algorithm (a timestamp includes both the year and the timestep). A handle for recording the **SystemState** at the start of iteration is also provided, along with some utility routines for manipulating this state. Three methods form the primary interface for this class. **iterationRequired()**, tests whether or not iteration is required given the current state of the system. A default definition for this method is provided which calls the pure virtual method **endConditionSatisfied()**, where the actual work of evaluating the end condition is performed in the derived class. **resetControlVariables()** is used to calculate new values for the iteration control variables at the start of each new iterative loop.

At present the only **IterationControl** derived class available is **MultiCeilingIterationControl**. An object of this class is used to implement the harvest management policy in which a single fishery harvests a fixed quota over multiple timesteps while maintaining a constant harvest rate over those timesteps. Many different objects of this class may all operate simultaneously in order to achieve this management objective for several fisheries in the same time period.

---

[4] Gamma, Helm, Johnson, Vlissides, “Design Patterns”, pp. 163-173

[5] Meyers, “More Effective C++”, pp. 183-213

5.5.2.5.3 IterationManager

The iteration system is controlled by the monostate **IterationManager**. This class organizes all of the **IterationControl** derived class objects. It ensures that for each,

- the system state is recorded and stored (in the **IterationControl** object) upon entry into the first timestep of interest for that control;
- that each **IterationControl** object evaluates its end condition at the completion of the last timestep of interest; and
- that iteration actually occurs when necessary, including resetting the system clock and causing **IterationControl** objects to reset their control variables when appropriate.

There is an implied ordering of **IterationControl** objects in which those with more recent "first" timesteps iterate to completion before those with earlier "first" steps. The **IterationManager** enforces this ordering by using a somewhat complicated internal ordered list implementation to contain the registered **IterationControl** objects.

In order to accomplish its tasks, the **IterationManager** must receive two messages at particular points during the simulation. At the start of each timestep, **beginTimeStep()** must be called, while **endTimeStep()** must be called at the end of each timestep. These calls are found in **RunTheModel()** in *engine.cpp*.

5.5.2.6 DataRequest

During the course of the simulation various data may be tracked, cumulated, or analyzed. For maximum flexibility, this is performed through the use of **DataRequest** objects. This abstract base class provides four methods as the primary portion of its interface: **config(), collect(), output(),** and **clear**(). The canonical usage is to register (or, more accurately, **schedule**()) any desired **DataRequest** objects with the **DataRequestManager**. This monostate ensures that the **collect()** methods are called for each **DataRequest** after every Simulation Process. Similarly, each of the **output()** methods is called after the simulation is complete.

Although the primary purpose of **DataRequest** objects is to collect output data, they may be used for other purposes as well. For example, the **Historian** monostate schedules a **TrackDetailedFisheryCatches** object (derived from **DataRequest**) with the **DataRequestManager** in order to efficiently collect a specific type of harvest data and make it available to any object during the course of the simulation. Certain harvest management algorithms make use of this summary data to analyze and implement management actions.

5.5.2.7 LogMsg

Warning and error messages are output to the user via the static function **LogMsg()**. This allows the programmer to output a printf style message to the screen (when the model is compiled as a console application.) Messages at various different "levels" may be created. Eventually this would allow the user to filter messages based on log level, although that functionality is not currently available. Additionally, all logged messages are copied to file *coast_log.txt*. The code for this module may be found in *log.h* and *log.cpp*.

5.5.3 Special Purpose Classes and Objects

The following sections describe a few of the classes and objects that participate in some of the more complicated operations in the model.

5.5.3.1 Harvest Classes

The most complex Simulation Process is harvest. The entry point for this process is **FisheryManager::takeHarvests()** (this call is found in *engine.cpp*), which is called to invoke the harvest process in each timestep**.** The workings of this method also provide an overview of the entire harvest process. The actions taken are:

1. Generate temporary data storage for the harvest mortality data. (This results in the creation of **FisheryUnit** and **HvMort** objects.)
2. Instruct the PolicyControlManager to perform preHarvestManagement().
3. For each Fishery:
    - Compute legal catches.
    - Compute shaker incidental mortality.
    - Compute CNR incidental mortality.
4. Instruct the PolicyControlManager to perform postHarvestManagement().
5. Update all cohort abundances by applying the computed harvest mortalities from each **Fishery**.

It is worthwhile to note that the organization of the harvest process presumes that within a timestep harvests for individual fisheries may be computed independently. The only way to achieve interactive effects across fisheries is by using the Iteration feature of the model (described above). Similarly, it is further assumed that harvest in a specific fishery and region (in a particular timestep) may be computed independently from other fisheries and/or regions. It is possible that this last requirement may be loosened at some point in the future, but for the moment the code structure adopts this as an invariant.

These sub-processes and the classes that implement them are described in further detail in subsequent sections.

5.5.3.1.1 FisheryUnit

Harvest actions occur independently in each region for each fishery. The **FisheryUnit** class encapsulates this intersection. At the start of the harvest process in each timestep, each fishery constructs a list of **FisheryUnit** objects representative of the regions in which harvest will occur for each fishery. This is accomplished via the method **Fishery::makeHarvestList()**. Each fishery is configured (from runtime data) with "schedule" information that specifies the times and regions in which it is to be active. This information is used to construct the proper **FisheryUnit** lists in each timestep.

Fisheries perform harvesting by sending a message instructing the **FisheryUnit** to do so. Each **FisheryUnit** contains a list of **HvMort** objects to contain the harvest mortality data. The **HvMort** class represents the intersection of a fishery, region, and cohort, and is responsible for the harvest calculation. A **FisheryUnit** will contain one **HvMort** object for each cohort to be harvested by the fishery in a particular region. Since cohorts are created and destroyed during the course of the simulation, this list of **HvMort** objects is also reconstructed at the start of each harvest process in each timestep, at the same time that the **FisheryUnit** lists are created. The actual work of creating the **HvMort** objects is performed by **HarvestManager::createHarvestMorts()** (called

by the **FisheryUnit**). The **FisheryUnit** retains ownership of these objects, and references are also handed to the fishery to ease certain accounting tasks. The **HarvestManager** considers whether the cohort is of harvestable age and has non-zero abundance in the region when deciding whether to create a corresponding **HvMort** object.

5.5.3.1.2 FisheryPolicy

FisheryPolicy is an abstract base class for conducting harvest under particular management policies. The interface includes two public methods: **harvest()** for performing all harvest in a single **Fishery** and **Region**; and **effortScalar()** to access the effort scalar used by that policy in its harvest computation. There are currently two concrete **FisheryPolicy** classes, **FixedHarvestRatePolicy** and **FixedQuotaPolicy**.

The **FixedHarvestRatePolicy** class performs harvest using a fixed harvest rate provided by input data (i.e. the effort scalar and base harvest rate do not change while executing this policy in a particular timestep).

The **FixedQuotaPolicy** enforces a quota upon the harvest for a single fishery in a single region in the current timestep. The quota may either be "forced" or not as specified by input data. An unforced quota means that harvest for a fishery will be truncated to the level of the quota, but harvest initially computed to be below the level of the quota is also allowed. A forced quota ensures that the final harvest of the fishery will meet the quota nearly exactly (within a certain tolerance, currently set at 0.001% of the quota). The **FixedQuotaPolicy** will select a new effort scalar and recompute harvest as necessary in order to meet the quota. This ability to search for the proper effort scalar is especially important if the harvest function itself is non-linear. The search is conducted using **Search** class object, which employs a guarded secant method (see inline comments in *Search.h* and *Search.*cpp for details).

It is required that there exist a **FisheryPolicy** object for every combination of Fishery, Region, and timestep in which harvest occurs. The **FisheryManager** maintains a **PolicyTable** for this data. At configuration time this table is initialized with default **FixedHarvestRatePolicy** objects with an effort scalar of 1.0 in each element. Input parameter data then overrides selected elements where other harvest management policies are desired.

5.5.3.1.3 PolicyControl

Some harvest management policies may span multiple timesteps. For example, the current PSC configuration of the model requires that fishery harvests be tracked over a base period before generating **FisheryQuotaPolicy** objects for the remainder of the simulation, using the base period average harvest as well as other scalars supplied as input data. The **PolicyControl** abstract class supports this type of functionality. **CeilingScalarPolicyControl** is the only derived **PolicyControl** class currently available. The interface of the **PolicyControl** class includes simply the **preHarvestManagementAction()** and **postHarvestManagementAction()** methods, called at the start and finish of the harvest process in each timestep, as described above. The **PolicyControlManager** monostate manages all the **PolicyControl** objects active in the system and ensures that each is called at the appropriate times.

5.5.3.1.4 HvMort and HarvestProcess

As described above, the **HvMort** class represents the intersection of a fishery, region, and cohort, and acts as a repository for the harvest mortality data (both legal and incidental) for that grouping. Its interface contains the **harvest()** method which takes the policy-determined effort scalar as a parameter in order to perform the harvest calculation.

Each **HvMort** object is configured with a **HarvestProcess** object used to finally calculate harvest mortality. The abstract **HarvestProcess** class contains a single method, **computeCatch()**. There is currently one **HarvestProcess** derived class, **LinearHarvestProcess**, which implements the simple formula:

```
        catch = cohort_abundance * (base_harvest_rate * (1 - pnv) * fp *
policy_effort_scalar)
```

The various parameters are obtained by instructing the **HarvestManager** to perform lookups into the data parameter generic array tables it maintains.

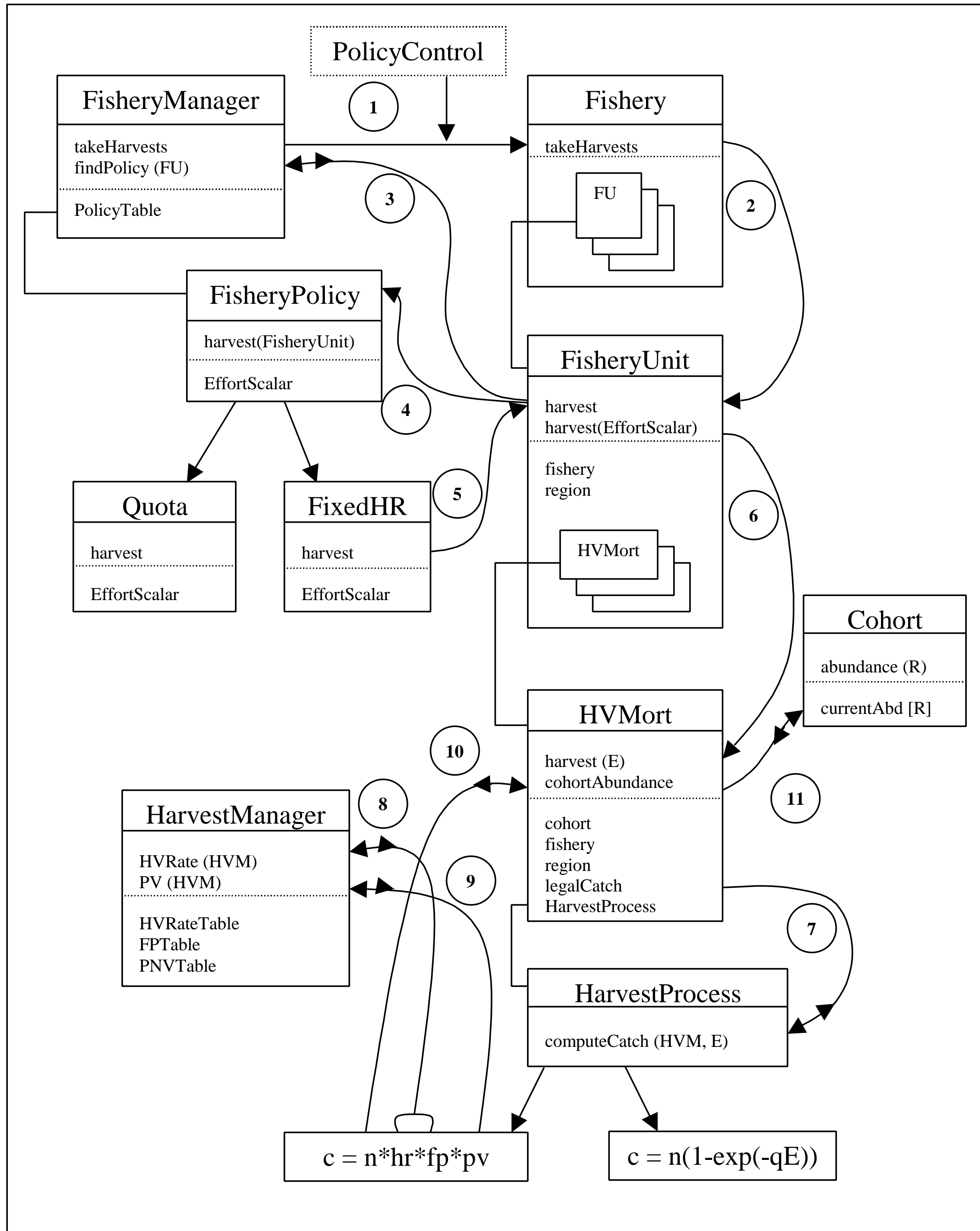
The **HvMort** class contains the following member variables, which serve as the repository for harvest mortality data: **legalCatch_, shakers_, dropOffs_, cnrSublegal_,** and **cnrLegal_**.

5.5.3.1.5 Legal Catch Process Detail

To summarize, using the classes and objects described in the preceding sections, legal catch is calculated as follows:

- The **FisheryManager** calls **Fishery::takeHarvests**() for each **Fishery**.
- The **Fishery** calls **FisheryUnit::harvest**() for each **FisheryUnit**.
- The **FisheryUnit** calls **FisheryManager::findPolicy(FisheryUnit*)** to retrieve the correct **FisheryPolicy** object.
- The **FisheryUnit** invokes the **harvest(FisheryUnit*)** method of the **FisheryPolicy** in order to allow the management policy to control the harvesting.
- The **FisheryPolicy** computes the proper effort scalar and invokes the **harvest(double effortScalar)** method of the **FisheryUnit**. The **FisheryPolicy** may repeat these actions as necessary until its management objectives are met (e.g. to enforce a quota.)
- The **FisheryUnit** calls the **harvest(double effortScalar)** method for each of its **HvMort** objects.
- The **HvMort** invokes the **computeCatch(const HvMort&, double policyScalar)** method of its contained **HarvestProcess** object.
- The **HarvestProcess** computes and returns a catch to the **HvMort** object, which saves this mortality data. This completes the computation of legal catch of a single cohort by a single fishery in a single region in a single timestep.

5.5.3.1.5.1 Harvest Computation Flow (Legal Catch)



5.5.3.1.6 Incidental Mortalities

As described in Section 5.5.3.1, after the **FisheryManager** computes legal catch for a fishery, it then proceeds to compute incidental mortalities for shakers and CNRs by invoking the **Fishery** methods **setShaker()** and **setCNR()**, respectively.

5.5.3.1.6.1 Shaker

The **Fishery** computes shakers for all of its harvests simultaneously. It retrieves from the **HarvestManager** the correct **Shaker** object. **Shaker** is an abstract base class which supplies the **setShakers()** method as part of its interface for this purpose. The result of this operation is that the **shakers_** and **dropOffs_** data elements of all the **HvMort** objects associated with the fishery will be set to new mortality values.

Concrete classes derived from **Shaker** that perform the computations in a variety of different ways (documented elsewhere) and are currently available are: **PSCShaker, PSCCustomShaker, PSCDropShaker,** and **PSCCustomDropShaker**.

5.5.3.1.6.2 CNR

The **Fishery** generates CNR mortalities by calling the **setCNR()** method of each of its **HvMort** objects. The **HvMort** then instructs the **HarvestManager** to return the correct **CNR** object. The **HvMort** then invokes the **sublegalMort()** and **legalMort()** methods of the **CNR** object, whose return values are used to set the **cnrSublegal_** and **cnrLegal_** member variables, respectively.

CNR is an abstract base class. The concrete classes derived from **CNR** that are currently available are: **CNR_HarvestRatio**, **CNR_HarvestRatioMultipleEncounter, CNR_ReportedEncounter, CNR_SeasonLength,** and **CNR_SeasonLengthMultipleEncounter.** The function of each of these classes is described elsewhere. This use of the **CNR** hierarchy is an implementation of the Strategy[6] design pattern.

5.5.3.2 Production Classes

The SpawningManager controls the Simulation Process that handles the production of fish. This monostate provides the process entry point **spawnCohorts()**. It also maintains a **ProductionTable** (generic array) of **Production** class objects. The basic procedure for spawning is fairly straightforward. Spawning is done at the stock level (as opposed to the cohort level). At each timestep, every stock and region are examined. If there is a **ProductionTable** entry for that stock and region (in the current timestep) then production will occur. The total adult mature regional abundance for that stock and region (regional escapement) is calculated, and the **spawn()** method of the appropriate **Production** object is invoked. The output of this method is the new abundance of age 0 fish for that stock and region. **CohortManager::makeNewAge0Cohorts()** is then called to create new active cohorts for the newly spawned fish. Some of the constituent classes in this process are described in further detail in the next sections.

5.5.3.2.1 Production and ProductionFunction

The **Production** class' primary function is to return a value (from the **Spawn()** method) representing the abundance of newly spawned fish from the given regional escapement. This is accomplished by maintaining a list of **ProductionFunction**s, and calling the **spawn()** method on each, supplying the full escapement value as argument, and totaling the results. **ProductionFunction** is an abstract base class. A typical **ProductionFunction** derived class will calculate the value of a particular production function using some

---

[6] Gamma, Helm, Johnson, Vlissides, "Design Patterns", pp.

portion of the total escapement, usually accomplished by enforcing the minimum and maximum escapement values of interest to that function. It is up to the user to ensure at data input time that the various minima and maxima for the group of **ProductionFunction**s in a given **Production** object combine to cover the full range of possible escapement values (unless implicit truncation is desired). For example, three functions might be configured such that the first would operate on the portion of the escapement abundance from 0 to 10,000 fish, while the next function would handle the portion of the escapement in the 10,000 to 15,000 range, while the last would handle all the remaining escaped fish above the level of 15,000 (if any).

At present, there are four available **ProductionFunction** derived classes, although more may be added in the future. These are described in the following table.

Table 4 Available ProductionFunction derived classes

LinearProduction	Simple linear function
RickerProduction	Ricker function
VariableTruncationRickerProduction	Ricker function with run-time variable truncation
EnhancedRickerProduction	Ricker function with enhancement and optional density dependence

5.5.3.2.2 CohortGenerator

After the abundance of new age 0 fish is calculated, the **CohortManager** is instructed to create new active cohorts via a call to **CohortManager::makeNewAge0Cohorts().** This in turn invokes **newAge0Cohorts()** on the stock in question, supplying the abundance, region, and brood year. Each **Stock** is configured with a **CohortGenerator** derived class, which performs the work of creating new age 0 cohorts. By default, a stock is configured with a **DefaultCohortGenerator** object. (At present, there is no user interface for selecting any cohort generator other than this default one.) The **DefaultCohortGenerator** simply creates a **CohortID** using the given stock and brood year, allowing other cohort characteristics to assume default values. This **CohortID** is used to search for currently active cohorts matching the id. If one is found, then the regional abundance of that cohort is incremented by the new abundance of age 0 fish. If a matching active cohort is not found, then a new one is created with the new regional abundance of age 0 fish set appropriately. The final effect is to create new age 0 fish for the current brood year in the region where they are spawned. It is expected that the age-specific transition matrices will be used to migrate those fish to the ocean at the appropriate time.

5.5.3.3 Data Output

As described above, data output for the model is handled by scheduling **DataRequest** objects with the **DataRequestManager**. At present there are three text output files generated in this manner. **OutputFisheryCatches** cumulates legal catch by year and fishery and outputs this data to file *stndcat.prn*. **OutputPSCStockEscapements** tracks escapement by stock and year and outputs this data to file *stndesc.prn*. Output file *coast_output.txt* contains Standard Data Sentence format output for the model run. The **StandardDataOutputManager** manages this file and generates the output data by scheduling DataRequest derived classes **SDS_Cabn, SDS_Nmrt,** and **SDS_Fmrt** with the **DataRequestManager**. The Standard Data Sentence format is described in Section 4. All of these **DataRequest** file output data objects are scheduled with the **DataRequestManager** during the initialization phase of the model.

5.5.3.4 Data Input Parser

The parser is a tool usable by other applications. In the Coast Model, it is employed by several of the initialization routines for the purpose of reading input data from text files. The input language for the Coast

Model is described Section 3. The fundamental abstract class for processing input file data is **Parser**. This class is derived from **FreeFormReader**, which processes sequential tokens from a file without formatting, while handling comments and special characters. Finally, this class contains a **LineBuffer** that is used to process a file a line at a time and extract tokens.

Parameter data is read by classes that derive from abstract base class **Parser**. **TopLevelParser** is one such class, responsible for processing tokens at the top level of the input hierarchy. Parameter data at other levels is handled by deriving from abstract class **SubParser,** which itself is derived from **Parser**. The purpose of **SubParser** is to handle the coordination of nested command block data (see Section 3). Examples of concrete classes deriving from **SubParser** include **ConfigParser**, **CohortParser**, **StockConfigParser**, and others.

GenArrayParser is another abstract class deriving from **SubParser**. It is used to process any parameter data which consists of a Generic Array (see Section 3 as well as descriptions of Generic Arrays in Section 5.5.2.2). Concrete classes deriving from **GenArrayParser** include **HarvestParser**, **FpParser**, **ProductionParser**, and many others.

Concrete classes deriving from **Parser** typically accomplish the work of processing tokens and calling corresponding class methods by using template class **KeywordProcessor**.

5.5.3.4.1 HarvestParser – An In-depth Example

This section examines the **HarvestParser** class in detail as an example of how to build modules to parse parameter data. The reader should first examine all the inline comments in the **Parser** and derived classes, including **GenArrayParser**, before attempting to follow this example. The reader should also be familiar with the **GenericArray** and **GenericArrayIndex** classes, with the coast model input language (described in Section 3), and in particular with the format for specifying generic array data in the input files.

5.5.3.4.1.1 HarvestParser Declarations

Basic harvest rates reside in a generic array table **HvRateTable** in the **HarvestManager** class. It consists of a generic array of double. A programmer wishing to write code to process input data for this generic array would begin by creating the **HarvestParser** class and making it a friend of the **HarvestManager**. Examining **HarvestParser** (the following code is found in *HarvestParser.h*):

```
// the harvest parser is a generic array parser

class HarvestParser : public GenArrayParser {
```

The purpose of this class is to parse data for a generic array. Therefore, it derives from **GenArrayParser** in order to parse all of the generic array tokens.

```
public:
  HarvestParser(Parser& p, RWCString dimension);
```

The constructor takes as arguments a parser with a currently open file, and the token that represents the dimension of the generic array requested by the user in the file.

```
protected:
  virtual void processKeyword(const RWCString& token)
     { keymap_.processKeyword(*this, token); }
```

The only public methods are those specified in the base classes. The virtual method **processKeyword()** is the canonical way to invoke processing functions on tokens. The member variable **keymap_** is the mapping between valid tokens and the corresponding class methods.

```
private:
    friend class KeywordProcessor<HarvestParser>;
```

```
        KeywordProcessor<HarvestParser> keymap_;
```

The mapping between tokens and methods is defined using template class **KeywordProcessor<HarvestParser>**, which must also be given friendship privileges.

```
        const RWCString dimension_; // string rep of dimensionality
```

The dimensionality of the generic array as given in the constructor is retained for future use.

```
        virtual void initKeymap();
        virtual void initState();
        void configHvRate(State* state);
```

These three initialization methods are all used by the constructor. **initKeymap()** sets up the token/method mapping. **initState()** initializes the state stack in the **GenArrayParser** portion of the class, necessary for processing nested layers of generic array tokens. **configHvRate()** sets up the actual generic array used to store the input data.

```
        void hvRate();
    };
```

There is only one simple token for the harvest rate data processed by this class. **hvRate()** is the method used to process the final harvest rate data when specified in the input file. This method is entered in the keymap with its corresponding token.

5.5.3.4.1.2 HarvestParser Definitions

This example is continued by examining the method definitions for the **HarvestParser** class, found in file *HarvestParser.cpp*.

```
    // ctor utility fns
    void
    HarvestParser::initKeymap()
    {
        keymap_.insert(HarvestTokens::HVRATE, hvRate);
```

The single simple token for this class is inserted into the keymap along with the address of the appropriate processing method. The actual string value of **HarvestTokens::HVRATE** happens to be "BaseHR", and is found in file *HarvestTokens.cpp*. This string is what appears in the input data file when the user specifies final base harvest rate values.

```
        initGenArrayKeymap(keymap_);
    }
```

All of the common tokens for processing generic arrays and their corresponding methods are also inserted into the keymap by calling this **GenArrayParser** base class method.

```
    void
    HarvestParser::initState()
    {
        // configure the initial state with the correct id list.
        State* state = new State;
```

The **State** class is declared in the **GenArrayParser** base class, and is used to handle the processing of nested levels of generic array specifications. (Note that this is a contained class in **GenArrayParser** for use in that class and classes derived from it. This is not the same as the system-wide **State** class described in Section 5.5.2.5.1.) Here a new state is created for the **HarvestParser** class to do its work. This **state** object is deleted in method **GenArrayParser::parseEnd()** when the final “end” token for this generic array is processed.

```
        state->context = "HarvestData";
```

A string describing the context of the current state is supplied for user messages.

```
        configHvRate(state);
```

This class method is called to create the generic array storage, and to configure the **state** object with the initial list of all of the **GenericArrayIndex** objects necessary to reference every possible element of this particular generic array.

```
        stateStack.push(state);
    }
```

Finally the properly configured **state** object is pushed onto the **stateStack** maintained by the **GenArrayParser** base class portion of this **HarvestParser**.

```
    HarvestParser::HarvestParser(Parser& p, RWCString dimension)
        : GenArrayParser(p), dimension_(dimension)
    {
        initState();
        initKeymap();
    }
```

The constructor initializes the base class, stores the dimension token, and calls the two initialization methods. All parsers derived from **GenArrayParser** will have constructors similar or identical to this one.

```
    // create a generic array for harvest rate data of the requested
    // dimension size and configure the HarvestManager with it.  also add
    // the list of generic ids to index every element of the array to the
    // current state.

    void
    HarvestParser::configHvRate(State* state)
    {
        GenericArrayFactory<double> hvFactory;
```

GenericArrays are created with **GenericArrayFactory**.

```
        GenericArrayFactory<double>::GenericArrayType arraytype =
                hvFactory.genericArrayType(dimension_);
```

Use the factory to identify the array type requested by the user with the token saved in **dimension_**.

```
        if (arraytype != hvFactory.NTypes) {
```

This tests whether the dimension specified by the user is valid.

```
        HvRateTable* hvRateTable =
             hvFactory.newGenericArrayFn1(arraytype,
             HarvestManager::defaultHvRate_);
```

Create the actual generic array using the factory and the default harvest rate as retrieved from the **HarvestManager**.

```
             HarvestManager::configHvRates(auto_ptr<HvRateTable>(
             hvRateTable)); // hand off harvest storage to the manager
```

As the comment indicates, this hands off the new **GenericArray** to the **HarvestManager**, which is where the final data is stored.

```
             hvRateTable->newGenericIndexList(state->ids); //
                        create the proper list of generic ids
        }
```

Create a list of **GenericArrayIndex** objects, one to reference every possible element in the generic array, and insert the items into the list at the top of the **State** stack.

```
            else {
                // unidentified dimension.  log an error, but allow this
                // parser to continue with an empty id list in the state, so
                // as to skip as much data as possible without generating
                // further messages.
                logError(pError(ParseError::BadData,
                    dimension_ + " is not a valid configuration for"
                    + state->context + "; discarding data")
                    .errorMessage());
            }
        }
```

Use the global error logger to generate an error message if the user requested generic array dimension was invalid.

```
        // read in one harvest object and place it in the HarvestManager
        // at every generic id region in the current list
        void
        HarvestParser::hvRate()
        {
            double rate;
            parseVar(rate);
```

This routine has been called through the keymap processor in response to the final "BaseHR" token. The parent class **parseVar()** method reads the actual data from the file, or throws an exception if the data is not of the proper type.

```
        // iterate through the ids in the current state
            RWTPtrOrderedVector<GenericArrayIndex>::iterator iter =
               (stateStack.top()->ids).begin();
            while (iter != (stateStack.top()->ids).end()) {
```

At this point any number of generic array tokens may have been processed, each of which reduced the list of **GenericArrayIndex** objects to the subset desired by the user. Now the data just read is to be assigned to all of the indices currently being referenced. The list is kept by the top object in the **stateStack** in the **GenArrayParser** portion of this class. The above lines of code set up an iterator to visit each of these **GenericArrayIndex** objects.

```
                GenericArrayIndex* id = *iter;
```

Dereference the iterator to obtain the **GenericArrayIndex** object.

```
        // hand off the ptr to the migration manager
                HarvestManager::insertHvRate(rate, *id);
                ++iter;
```

Insert the harvest rate data item at the index by handing it to the **HarvestManager**. Increment the iterator.

```
        }
            // pop one level of state after processing the data
            parseEnd();
        }
```

All generic array blocks pop one level of state after parsing the final simple data token. This has the effect of subsuming the "end" token that would normally be required at every generic array level in the data file. See the coast model input language description in Section 3 for more details.

5.5.3.4.1.3 Using the HarvestParser

All that remains is to create a token for the generic array that will be created by the HarvestParser and to use it in an existing parser. File *TopTokens.cpp* contains the following definition:

```
const RWCString TopTokens::HVRATE = "HarvestRateData";
```

This is declared in *TopTokens.h*. A new method is created in the **TopLevelParser** class, **harvestData()**. This method is connected with the new token in the keymap in the **TopLevelParser**:

```
TopLevelParser::
TopLevelParser(const RWCString& fileName)
: Parser(fileName, &::errorLogObject)
{
//… other tokens are also inserted into the keymap here
    keymap_.insert(TopTokens::HVRATE, harvestData);
//…
```

The new **harvestData** method will be invoked by the keymap when the new token is encountered in the data file, at which time a **HarvestParser** object will be created to continue processing the file, as shown below.

```
void
TopLevelParser::harvestData()
{
    RWCString dimension = nextToken();
    HarvestParser hp(*this, dimension);
    hp.parse();
}
```

The dimension token is read following the "HarvestRateData" token and passed to the **HarvestParser**, whose virtual **parse()** method is then called. Upon completion, the **HarvestParser** is discarded and file processing continues in the **TopLevelParser** with the next token.

5.6 Process Detail

This section provides further information on two fundamental processes in the application which are not controlled by system monostates or other classes.

5.6.1 Initialization and Cleanup

Initialization code is located in file *init.cpp*. Initialization consists of three distinct steps. **::ModelConfig()** executes each of these steps in turn. First is configuration. In this phase the configuration data file is read (*config.data*), after which the static and global constructs are set up, typically by calling **config()** methods in each of the managers. Next the general model data file is read (*coast.data*) in routine **::ModelDataRead()** which establishes the parameter data for the model from the user input files. Finally, **::ModelInit** performs per-simulation initializations as final preparation for the model run. For calibration runs a separate entry point, **::activeXcalibrationReentry()** is provided for iterations subsequent to the first one. This is a speed optimization, in which the full original data file is not reread. In this case **::ModelConfig()** is not called. Instead, a supplemental data file (*calib.data*) is read using **::ModelDataRead()**, which allows the user to overwrite only the new calibration input data (typically ev scalars) and retain the remainder of the original data set. **::ModelInit()** is then called as usual. Some peculiarities may limit the use of this optimization for configurations other than a PSC calibration run. Please see comments in the code for more details.

There are two phases to cleanup. **::SimulationWrapup()** occurs after every simulation run. **::ModelCleanup()** is the final cleanup routine called on exit. In an ActiveX application, the former is called at

the end of each simulation run, while the latter is not called until the component itself exits (which occurs when the calling application exits). Both of these routines are found in *init.cpp*.

5.6.2 The Engine

The simulation is controlled by routine **RunTheModel()** in file *engine.cpp*. It loops through years and timesteps, within which each of the Simulation Processes are executed through a call to the appropriate manager. The **IterationManager** is informed of the beginning and ending of each timestep, and after each Simulation Process, the **DataRequestManager** is afforded the opportunity to store any requested data. The entire routine is reproduced below.

```
/* main engine for the nmfs coastal simulation.  */

void RunTheModel()
{
    for (::SystemClock->resetClock(); !(::SystemClock->yearExpired());
         ::SystemClock->incrementYear()) {
         ::LogMsg(L_Msg, "year %d\n", ::SystemClock->currentCalendarYear());

        for (::SystemClock->resetTimeSteps(); !(::SystemClock->timeExpired());
             ::SystemClock->incrementTimeStep()) {

            // the IterationManager could reset the SystemClock to an earlier time
            IterationManager::beginTimeStep();

            // aging up to the current year/time needs to happen before any data
collection
            CohortManager::ageCohorts();

            DataRequestManager::collect(SystemProcesses::SP_BeforeFirstProcess);

            NaturalMortalityManager::takeNaturalMortality();
            DataRequestManager::collect(SystemProcesses::SP_NaturalMortality);

            FisheryManager::takeHarvests();
            DataRequestManager::collect(SystemProcesses::SP_Harvest);

            MaturationManager::maturateCohorts();
            DataRequestManager::collect(SystemProcesses::SP_Maturation);

            SpawningManager::spawnCohorts();
            DataRequestManager::collect(SystemProcesses::SP_Spawn);

            MigrationManager::migrateCohorts();
            DataRequestManager::collect(SystemProcesses::SP_Migration);

            IterationManager::endTimeStep();
        }
    }
}
```

Appendix A: Glossary

The following is a partial glossary of terms used in the Coast Model.

Term	Definition
Abundance Index	The expected catch given the current year size limits and cohort sizes but the base period (1979-1981) harvest rates.
Adult Equivalence Factors	Used to adjust fishery catches to a common impact on the spawning stock. For example, on average a three year old fish harvested by an ocean fishery has less impact on the spawning stock than a five year old fish harvested by a river fishery, because some three year old fish would normally die of natural causes before they had an opportunity to spawn. Thus, one three year old fish eliminated from the ocean catch will result in less than one additional fish in the spawning stock, whereas one five year old fish eliminated from the river catch will result in one additional fish in the spawning stock.
Adult Escapement	Terminal Run fish that survive the terminal fisheries and pre-spawning mortality. Age two fish returning to the river are not considered reproductively viable and are not included in the adult escapement for each stock.
Base Period Harvest Rate	Average stock, age, and fishery specific harvest rate between 1979-1982. Harvest Rate scalars are relative to this rate.
Brood Year	The year in which a fish was propagated or spawned (i.e., the year in which the eggs were fertilized). Chinook salmon typically migrate downstream the following year (most Fall chinook), or the year after (most Spring chinook).
Catch Ceilings	Maximum catch (numbers of fish) for a fishery or group of fisheries for a specified time period. These are not established for specific stocks. This is the Pacific Salmon Commission's primary management tool.
Chinook Non-Retention Mortalities	Mortalities of legal and sub-legal chinook that are caught and brought up to the boat in coho fisheries at times when it is not legal to land and sell any chinook.
CNR (mortalities)	See Chinook Non-Retention Mortalities.
Coded-Wire-Tag (CWT)	Tiny wire tags (1.0 x 0.25 mm) inserted in the nose cartilage of salmon fingerlings or fry, typically in the hatchery, to identify the origin of an individual fish. Each tagged fish has the adipose fin clipped to indicate that it has a CWT in its snout. Scientists use CWT recoveries to estimate harvest rates and migration patterns.
Cohort	A group of fish that have the same demographic characteristics, such as belonging to the same age class of a given stock.
Cohort Analysis	Same as Virtual Population Analysis.
Enhancement	Production of fish at facilities such as hatcheries.
Escapement	Fish that are not caught by any fisheries (i.e., they "escape" the fisheries).
EV Scalar	Scalars used to adjust the average production of age one fish by a spawning stock to account for inter-annual Environmental Variability (EV).

Term	Definition
Gillnet	A harvest method in which fish are trapped in a net stretched across their migration path. The net may either be set from a drifting boat (drift gillnetting) or from a fixed position (set gillnetting). The fish become entangled by their gill plates or jaws, and can neither back out nor move forward.
Harvest Rate Scalars	Scalars used to adjust the harvest rate during a given year compared to the Base Period.
IDL (rate)	See Inter-Dam Loss rate.
Inter-Dam Loss rate	These are actually *survival* rates between the last fishery and the spawning grounds. Also called the Pre-spawning mortality. IDLs are stock specific, but are not age (or size) specific. This mortality is applied to Columbia River stocks that spawn upriver from dams and is assessed after fishing mortality to account for losses between dams.
Legal (size)	Above a certain size criteria.
Maturation Rates	The proportion of a stock that is mature and ready to return to the spawning ground. These are age and stock specific and can vary across years as well. However the model does not allow for age 6 fish so the MR for age 5 fish should always be 1. The stock that is mature is considered the terminal run.
Natural Ocean Mortality	Non-fishing mortality assessed at the beginning of each year in the model. This mortality is age specific, but not stock specific.
Net Fisheries	In CRiSP Harvest, this refers to fisheries using gillnet and purse seine gears.
Pacific Salmon Commission	International regulatory agency created by the 1985 Pacific Salmon Treaty between the United States and Canada with responsibility for management of North American salmon stocks and fisheries.
Percent Non-Vulnerable	Fraction of a cohort that is below the legal size limit. PNVs vary by year, age, and fishery, but not by stock.
PNV	See Percent Non-Vulnerable.
Pre-Spawning Mortality	See Inter-Dam Loss.
Preterminal (catch)	Catch that occurs before the mature segment of a cohort begins migrating back to the spawning grounds. Thus, preterminal catches are primarily ocean catches.
PSC	See Pacific Salmon Commission.
Purse Seine	A commercial fishing system in which a school of fish are encircled by a vertically hanging net and then are trapped by closing the bottom of the net (pursing).
Recruitment	Fish from a given stock that become available (i.e., recruit) to a fishery.
Recruitment Age	The age at which fish from a given stock become available to a fishery.
Ricker Function	A popular type of Spawner/Recruit Relationship (named after Dr. William Ricker) in which the number of recruits per spawner declines exponentially. The resulting curve has a descending right hand limb (i.e., too many spawners produce fewer recruits).
Shakers	Sublegal chinook that are caught (i.e., hooked and brought up to the boat) and released (i.e., "shaken" off the gear) during directed chinook fisheries.

Term	Definition
Spawner/Recruit Relationship	A mathematical relationship between the number of spawners in a given year and the resulting number of progeny that become available (i.e., recruit) to the fisheries in some future year. Usually estimated from historical data and used in simulation models to predict future recruitment from a given spawning stock.
Sub-legal (size)	Below a certain size criteria.
Supplementation	Artificial propagation intended to reestablish or increase the abundance of natural populations.
Terminal Catch	Catch of the mature segment of a cohort as it migrates back to the spawning grounds. Some ocean net catches that occur in nearshore waters are considered terminal catches.
Terminal Run	Mature fish leaving the open ocean and returning to the spawning grounds. Compare to True Terminal Run.
Total Catch	Sum of the Preterminal and Terminal catches.
Troll	A commercial harvest method for chinook and coho salmon, usually in the open ocean, that captures individual fish on lures or baited hooks being slowly pulled through the water.
True Terminal Run	The Terminal Run minus nearshore ocean net catches. Thus, it is the number of fish entering the natal river (as opposed to the number of mature fish leaving the ocean feeding areas). Compare to Terminal Run.
Virtual Population Analysis	A technique (sometime referred to as VPA) for reconstructing the history of a cohort of fish. By counting the number of spawners and the catches and making estimates of the natural mortalities it is possible to reconstruct the history of a cohort.

Appendix B: Examples

B.1 Introduction

We have developed several examples of Coast Model configurations to illustrate how the code can be used to build progressively more complex models. The distribution includes the Coast Model executable (coast.exe) and separate folders containing the input files for each prototype. However, since the data and executable must be in the same directory to run the model, you should create a shortcut for coast.exe in each prototype folder that you wish to run. The following steps describe how to create this shortcut:

1. Drag the "coast.exe" icon into one or more prototype folders as desired. This creates the shortcut.
2. Within each prototype folder, set the executable link to start in the prototype folder, as follows:
    - Right click on “Shortcut to coast.exe” to bring up a menu list;
    - Select **Properties**;
    - Click on the **Shortcut** tab;
    - Type in the complete path to the prototype folder in the **Start In** text box;
    - Click **OK**.

To run the model, double click on “Shortcut to coast.exe” within each prototype folder.

When the simulation is complete, the program will generate three files:

- coast_output.txt (standard data sentences)
- stndcat.prn (catches by fishery and year)
- stndesc.prn (escapements by stock and year)

The coast_output.txt file can be quite large. If the distribution is from a floppy disk, do not attempt to run any of the examples from the floppy disk-the resulting output file will be too large to save on the disk.

The examples which follow demonstrate the flexibility of the Coast Model with respect to the number of Stocks, Fisheries, Regions and Timesteps. The model itself can handle a wide range configurations for any of these parameters simply by changing the input data files. Considerations which must be weighed when changing these parameters are more closely related to the nature of the related data. For example, Transition Matrices are specified by Region and Timestep. Thus, if the number of Timesteps or Regions is increased, all of the **TransitionMatrix** data (*“.tm”* file) must be modified to reflect this change. The same holds true for Maturation data (“*.mat”* file). Base harvest rates (*“.bhr”* file) are specified by Fishery, Region, Timestep, Stock, and Age, so a change to any of these must be addressed in that data. Some of the relationships between the various possible **GenericArray** dimensions and the bulk of the input data are rather subtle, so care should be taken to understand the nature of all of the data and algorithms in the model when making these changes. A careful study of the following examples and a comparison of the data files which implement them can provide a great deal of insight into this issue.

B.2 Prototypes With No Harvest

These prototypes illustrate how to build a biological system (no harvesting) of increasing complexity, or resolution. Proto 0 shows how the code can be used to plot a stochastic Ricker function. Prototypes 1 through 5 all have a single stock with the same production function, but increasing time/space resolution. The Stndesc.prn files for each of these prototypes are identical, even though the biological processes within each year are simulated differently. Prototypes 6 and 7 include both chinook and coho stocks.

B.2.1 Proto 0 (One chinook stock over 100 years)

- Stocks: 1 chinook.
- Regions: 2 (R1:R2).
- Timesteps: 2 (T1:T2).
- Natural Mortality: Only in T1, R1.
- Harvest: None.
- Maturation: Only at end of T1.
- Spawning: Only in T2, R2; Ricker production with random EV Scalars for each year.
- Migration: All mature fish migrate to R2 at end of T1.

B.2.2 Proto 1 (Like PSC chinook model, but only one stock)

- Stocks: 1 chinook.
- Regions: 4 (R1:R4).
- Timesteps: 4 (T1:T4).
- Natural Mortality: Only in T1, R1.
- Harvest: None.
- Maturation: Only at end of T1.
- Spawning: Only in T4, R4.
- Migration: PSC style.

B.2.3 Proto 1a (Like PSC chinook model, but simulating one coho stock)

- Stocks: 1 coho.
- Regions: 4 (R1:R4).
- Timesteps: 4 (T1:T4).
- Natural Mortality: Only in T1, R1.
- Harvest: None.
- Maturation: Only at end of T1; all age 3 fish 100% maturation.
- Spawning: Only in T4, R4.
- Migration: PSC style.

B.2.4 Proto 2 (Add "estuary" and more ocean regions; stock distribution by ocean region)

- Stocks: 1 chinook.
- Regions: 7 (R1 = Estuary; R2:R4 = ocean; R5 = terminal; R6 = river; R7 = spawning).
- Timesteps: 5 (T1:T5); T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs only in T2 in regions R1:R5.
- Harvest: None.
- Maturation: Only occurs at end of T2 in regions R2:R4.
- Spawning: Only in R7, T5.
- Migration: Migration at end of first step distributes immature fish from the estuary to the ocean regions. Mature fish move to R5 after T2, R6 after T3, and R7 after T4.

B.2.5 Proto 3 (Add more terminal regions; stock distribution by region)

- Stocks: 1 chinook.
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 5 (T1:T5); T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Only in T2 in all regions R1:R9.
- Harvest: None.
- Maturation: Only occurs at end of timestep 2 in ocean regions R2:R4.
- Spawning: Only in R9, T5.
- Migration: Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). Mature fish move to terminal regions (R5:R7) after T2, to R8 after T3, and R9 after T4.

B.2.6 Proto 4 (Increase timesteps to 13; spread natural mortality over timesteps; migration during most timesteps)

- Stocks: 1 chinook.
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 13 (T1:T13); T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs equally over T2:T6 in all regions R1:R9.
- Harvest: None.
- Maturation: Only occurs at end of T6 in ocean regions R2:R4.
- Spawning: Only in R9, T13.
- Migration: Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). During timesteps T2:T13 immature fish move around the ocean areas. Mature fish move to terminal regions (R5:R7) after T6 and then through the terminal and river areas during T7:T12, winding up in the spawning area in T13.

B.2.7 Proto 5 (Maturation occurs during timesteps 4 to 7 in all ocean regions)

- Stocks: 1 chinook.
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 13 (T1:T13); T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs equally over timesteps T1:T12 and all regions R1:R9.
- Harvest: None.
- Maturation: Occurs during T4:T7 in all regions R1:R9.
- Spawning: Only in R9, T13.
- Migration: Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). During timesteps T2:T13 immature fish move around the ocean areas. During T5:T12 mature fish move systematically from the ocean regions to the spawning region.

B.2.8 Proto 6 (same as Proto 5, but add the coho stock)

- Stocks: 2 (1 chinook; 1 coho).
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 13 (T1:T13); T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs equally over timesteps T1:T12 and all regions R1:R9.
- Harvest: None.
- Maturation: Occurs during T4:T7 in all regions R1:R9; coho stock has cumulative 100% maturation rate (over T4:T7) for age 3 fish.
- Spawning: Only in R9, T13.
- Migration: Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). During timesteps T2:T13 immature fish move around the ocean areas. During T4:T12 mature fish move systematically from the ocean regions to the spawning region.

B.2.9 Proto 7 (each stock different natural mortality, maturation, and migration process)

- Stocks: 1 chinook; 1 coho.
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 13; T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs equally over timesteps T1:T12 and all regions R1:R9, but different values for each stock.
- Harvest: None.
- Maturation: Occurs during T4:T7 in all regions R1:R9; coho stock has cumulative 100% maturation rate (over T6:T9) for age 3 fish.
- Spawning: Only in R9, T13.
- Migration: Different parameters for each stock. Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). During timesteps T2:T13 immature fish move around the ocean areas. For chinook stock during T4:T12 mature fish move systematically from the ocean regions to the spawning region. For coho stock during T7:T12 mature fish move systematically from the ocean regions to the spawning region.

B.3 Prototypes With Harvest

These examples superimpose fishing processes over the biological system of Proto 7. Proto 10 illustrates how the model can handle catch ceilings for fisheries operating in different regions and during different timesteps.

B.3.1 Proto 8 (Same as Proto 7, but add an ocean fishery)

- Stocks: 1 chinook; 1 coho.
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 13; T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs equally over timesteps T1:T12 and all regions R1:R9, but different values for each stock.
- Harvest: F4 (WCVI ocean troll fishery) operates in regions R2:R4 during timesteps T5:T6.
- Maturation: Occurs during T4:T7 in all regions R1:R9; coho stock has cumulative 100% maturation rate (over T6:T9) for age 3 fish.
- Spawning: Only in R9, T13.
- Migration: Different parameters for each stock. Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). During timesteps T2:T13 immature fish move around the ocean areas. For chinook stock during T4:T12 mature fish move systematically from the ocean regions to the spawning region. For coho stock during T7:T12 mature fish move systematically from the ocean regions to the spawning region.

B.3.2 Proto 9 (Same as Proto 8, but add an inside fishery that has a multi-phase catch ceiling that spans three timesteps)

- Stocks: 1 chinook; 1 coho.
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 13; T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs equally over timesteps T1:T12 and all regions R1:R9, but different values for each stock.
- Harvest: F4 (WCVI ocean troll fishery) operates in regions R2:R4 during timesteps T5:T6; F13 (Puget Sound South Net fishery) operates in regions R5:R7 during timesteps T7:T9 and has annual catch ceilings.
- Maturation: Occurs during T4:T7 in all regions R1:R9; coho stock has cumulative 100% maturation rate (over T6:T9) for age 3 fish.
- Spawning: Only in R9, T13.
- Migration: Different parameters for each stock. Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). During timesteps T2:T13 immature fish move around the ocean areas. For chinook stock during T4:T12 mature fish move systematically from the ocean regions to the spawning region. For coho stock during T7:T12 mature fish move systematically from the ocean regions to the spawning region.

B.3.3 Proto 10 (Same as Proto 8, but add a third fishery; all fisheries have multi-phase catch ceilings spanning five timesteps)

- Stocks: 1 chinook; 1 coho.
- Regions: 9 (R1 = estuary; R2:R4 = ocean; R5:R7 = terminal; R8 = river; R9 = spawning).
- Timesteps: 13; T1 used to distribute fish from estuary (R1) to ocean regions (R2:R4).
- Natural Mortality: Occurs equally over timesteps T1:T12 and all regions R1:R9, but different values for each stock.
- Harvest: F4 (WCVI ocean troll fishery) operates in regions R2:R4 during timesteps T5:T7; F11 (Juan de Fuca Net fishery) operates in region 5 during timesteps T6:T9; F13 (Puget Sound South Net fishery) operates in regions R6:R7 during timesteps T8:T10 (see Proto10.bhr for all fisheries); all fisheries have annual catch ceilings defined over timesteps T5:T10 (see Proto10.cei).
- Maturation: Occurs during T4:T7 in all regions R1:R9; coho stock has cumulative 100% maturation rate (over T6:T9) for age 3 fish.
- Spawning: Only in R9, T13.
- Migration: Different parameters for each stock. Migration at end of first step distributes immature fish from the estuary to the ocean regions (R2:R4). During timesteps T2:T13 immature fish move around the ocean areas. For chinook stock during T4:T12 mature fish move systematically from the ocean regions to the spawning region. For coho stock during T7:T12 mature fish move systematically from the ocean regions to the spawning region.

Appendix C: Discussion Papers

C.1 Overview

Throughout the course of this project there were many discussions about how best to model each process. These discussions were documented in the meeting minutes and other discussion papers. To provide background information about why the Coast Model is designed the way it is, the following sections summarize these discussions. We hope that future programmers and modelers find these discussions helpful.

C.2 Ageing Process

C.2.1 Spring Stock Algorithms

Here's the sequence of events for spring stocks. The initial abundances for ages 2:5 are contained in *.stk. These are read in and then stored in a random access file. The age 1 initial abundance is computed using the Scale2To1 factor as for the fall stocks. The new wrinkle is that a new abundance called SprAge1 (effectively an age 0 abundance) is created and set equal to the initial Age 1 abundance. Here's the QB code (in Chinpt8.bas):

```
'..... Initial cohort sizes
        '***** 2/96 save FirstCohort data as random access file ******
        'FOR age% = 2 TO NumAges
        '   INPUT #3, FirstCohort(age%, Stk%)
        'NEXT age%
        'FirstCohort(1, Stk%) = FirstCohort(2, Stk%) * Scale2TO1
        '**************** 1/96 SPRING STOCK PROVISION ********
        'Spring(Stk%) = FirstCohort(1, Stk%)
        '*******************************************************
        INPUT #3, CO.CohAge2
        INPUT #3, CO.CohAge3
        INPUT #3, CO.CohAge4
        INPUT #3, CO.CohAge5
        CO.CohAge1 = CO.CohAge2 * Scale2TO1
        CO.SprAge1 = CO.CohAge1
        PUT #COFile%, Stk%, CO
        '*******************************************************
```

Thus, there are essentially six initial cohort abundances created, with CO.SprAge1 being an age zero abundance. In module Chsim8.bas the Sub IResetIter retrieves the data from the random access file. The initial abundances are scaled by the EV scalars (called StkSclr in the code). Again, an age zero abundance is created by setting a variable called Spring(Stk%) equal to the age one abundance. Note that the CO.SprAge1 variable (assigned during data input) is not used. Here's the code:

```
'***** 2/96 ***** Get initial cohort sizes from random access file ****
         GET #COFile%, Stk%, CO
         Cohort(1, Stk%) = CO.CohAge1 * StkSclr(-1, Stk%)
         Cohort(2, Stk%) = CO.CohAge2 * StkSclr(-2, Stk%)
         Cohort(3, Stk%) = CO.CohAge3 * StkSclr(-3, Stk%)
         Cohort(4, Stk%) = CO.CohAge4 * StkSclr(-4, Stk%)
         Cohort(5, Stk%) = CO.CohAge5 * StkSclr(-5, Stk%)
         Spring(Stk%) = Cohort(1, Stk%)
```

The Cohort(1,Stk%) and Spring(Stk%) variables are not used again until its time to produce new fish for the next year. Here's the production code:

```
'..... Compute age 1 cohort size
'*********** 1/96 SPRING STOCK PROVISION **********************
        IF HatchFlg%(Stk%) > 1 THEN
             '..... This is a spring stock, delay recruitment by a year
              Cohort(1, Stk%) = Spring(Stk%) * StkSclr(Yr%, Stk%)
              Spring(Stk%) = Age1Fish
        ELSE
             '..... This is a fall stock
              Cohort(1, Stk%) = Age1Fish * StkSclr(Yr%, Stk%)
        END IF
        '********** END CHANGE ***************************************
```

Note that when the initial Age 0 spring stock cohort is finally brought into the model at the end of the first model year, it gets multiplied by two EV scalars. After algebra, the net effect is:

```
Cohort(1, Stk%) = CO.CohAge1 * StkSclr(-1, Stk%) * StkSclr(Yr%, Stk%)
```

Thus, the Age 0 cohorts for spring stocks must be multiplied by two EV Scalars in the Coast Model *.coh file.

In later years, the variable Age1Fish is computed normally using one of the production functions. However, note that this Age1Fish does not include the effects of the EV scalar, but we DO include EV scalars in our production function. This is not a problem if its a fall stock because the CTC code immediately applies the EV scalar to the Age1Fish when assigning the age to a cohort abundance.

If its a spring stock, the Age1Fish variable is assigned to the Spring(Stk%) variable (essentially the age 0 abundance) without applying the EV value. The EV value is applied the following year when the Spring(Stk%) variable is assigned to age 1 abundance. The net effect is that for spring stocks the EV scalar from year y+1 is applied to the fish generated in year y (as shown in the above code).

Thus, the Coast Model production functions will still work OK for spring stocks PROVIDED WE GET THE RIGHT EV APPLIED IN THE RIGHT YEAR. This can be accomplished by replacing EV(y) with EV(y+1) in the production function specified in the *.prd file. Algebraically we have

Age1Fish(y) = F(spawners(y))

Spring(y) = Age1Fish(y)

Cohort1(y) = Spring(y)*EV(y+1) = Age1Fish(y)*EV(y+1)

What happens in the last year (y = last year of model run)? It doesn't matter because the new production is never used. Soooo …. we just set EV(y+1) = 1 for the production function in the last year.

C.3 Mortality Processes

C.3.1 State Space Model Considerations

A critical assumption of this project was that Ken Newman's State Space Model (SSM) and Kalman filter estimation methodology would eventually be used to estimate natural mortality, harvest, and migration parameters for a new generation of management models. Because this methodology uses instantaneous mortality rates in the underlying sub-models, there was concern that the Coast Model code frame be consistent with those equations. The following discussion paper on this subject was posted by Jim Norris on the web site on July 23, 1998 and was discussed at the July 30, 1998 committee meeting.

C.3.1.1 Background

At the July 2, 1998 meeting Jim Scott asked whether or not harvest parameter estimates from the State Space Model (SSM) would be compatible with our proposed Harvest Process and Fishing Process concepts. Before discussing compatibility, I summarize these concepts. Our purpose in rigorously defining these concepts is to clearly identify the inputs, outputs, data requirements, and functions of these critical code objects.

C.3.1.2 Harvest Process

Within a given year, timestep, region, and fishery a harvest process defines the interaction between the amount of fishing effort (i.e., number of people involved) and the number of fish from a given stock and cohort. In this context we define a "fishery" to include all regulations and properties other than the amount of fishing effort (e.g., size limits, bag limits, and selective fishery rules). A cohort is defined to be any group of fish having the same identifying characteristics and demographic features (e.g., parent stock, tag status, mark status, sex, growth group, and genetic group).

In virtually all types of fishery simulation models, there is a line of code (occasionally more than one line) that assigns a legal catch at the year, timestep, region, fishery, stock, and cohort level. In most cases, this line of code represents what we call a harvest process. Three common types of equations are the following:

1. Simple Linear Rate (used by FRAM & PSC chinook model).

```
C(c,f) = HR(c,f)*N(c)
```

where

C(c,f) = catch of cohort c in fishery f

HR(c,f) = harvest rate for cohort c in fishery f

N(c) = abundance of cohort c at start of period.

2. Non-Linear Relationship (similar to PM Model).

```
C(c,f) = (1 - exp(-q(c,f)*E(f)))*N(c)
```

where

q(c,f) = catchability coefficient for cohort c in fishery f

E(f) = effort in fishery f during period.

3. Instantaneous Rates (used by SSM).

```
F(c,f)
      C(c,f) = ------ * (1 - exp(-Z(c))) * N(c)
                 Z(c)
```

where

F(c,f) = instantaneous rate of fishing mort

Z(c) = instantaneous rate of total mortality for cohort c, and

Z(c) = M(c) + Sum[F(c,f)] over all f

F(c,f) = q(f)*E(f)

M(c) = instantaneous rate of natural mortality for cohort c.

C.3.1.3 Fishing Process

For each year, timestep, region, and fishery a fishing process defines the amount of fishing effort to be input into the harvest processes for all cohorts residing in the given time and region in order to satisfy some management objective. Note that under this formulation, a fishing process does not compute any fishing mortalities--it only determines the inputs to the harvest processes. Only harvest processes compute fishing mortalities. Note also that a fishing process applies only to a single year, timestep, region, and fishery. This is the issue I think Jim Scott was concerned about.

In the PSC Chinook Model, non-ceilinged fisheries have a fixed harvest rate management objective. Thus, the FPs are set for each fishery at config time and are passed into each harvest process without modification. On the other hand, each simple ceilinged fishery adjusts the effort level for all harvest processes in a given year, region, and timestep by a scalar (called the RT factor) in order to make the sum of the legal catches meet the management objective.

C.3.1.4 Code Issues

Up until now we had not considered implementing instantaneous rate equations. From a code perspective, instantaneous rate equations for harvest processes are fundamentally different from the linear and non-linear equations because all the required information is not autonomous within a single fishery. Specifically, the instantaneous natural mortality rate for each stock is needed, along with the instantaneous fishing mortality rates for all other fisheries within the same region. Thus, to implement instantaneous rate equations, a fishery object must have access to this outside information. If two fisheries operating in the same region at the same time both have quotas, whenever effort in one fishery is adjusted to meet its quota, the Z value (total instantaneous mortality rate) changes for all stocks. Thus, even simple quota fisheries will have to be solved together through a common algorithm. At a more fundamental level, the natural mortality and fishing mortality processes are intertwined and must be computed simultaneously. That is, one cannot compute natural mortalities and then move on to computing fishing mortalities , unless the fishing effort levels do not require adjustment to meet some objective. For all of the above reasons, I conclude that our proposed code structure is not compatible with using instantaneous rate equations.

C.3.1.5 Code Solution

First, we must combine the Natural Mortality process and the Fishing Mortality process into a single Mortality process during each timestep of a model run. A model can be configured to use one of two types of Mortality processes. One type can compute natural mortalities and fishing mortalities independently (as most models currently do). Or, in a second type, the natural and fishing mortalities can be computed simultaneously using instantaneous rates. At the start of the mortality process, all instantaneous rates would have to be determined. For example, the natural mortality rates could be related to the physical environment of the region and/or the average size (length) of the individual in the cohort. Likewise, the fishing mortality rates could be determined by fixed effort levels, or the effort levels for some fisheries could be set dynamically via some algorithm. The key point is that somehow all the rates are established prior to any computations. Once the rates are established, the natural mortality computations and fishing mortality computations can be made independently. If there are any constraints within the given timestep and region (e.g., quotas, escapement goals, allocations), then an algorithm must be written to adjust fishing effort levels at the start of the total mortality process level.

C.3.1.6 Code/Algorithm Problems

Finding effort levels to meet some constraints using instantaneous rate equations can be tricky. The fact that the total instantaneous mortality rate (Z) is the sum of several individual fishery rates can lead to an infinite number of acceptable solutions. For example, if there is an allocation goal to equalize the sum of all Treaty fisheries with all Non-Treaty fisheries within a region, there can be many solutions (unless allocations WITHIN the Treaty and Non-Treaty groups are also specified). I used an Excel spreadsheet with Solver to model four fisheries, four stocks, and two timesteps (using instantaneous rate equations), and found that one must be very specific about the constraints in order to have a unique solution. The bottom

line is that it is easy to make the model framework compatible with instantaneous rate equations, provided there is never any need to adjust fishing effort levels to meet management constraints. If constraints must be met, it looks like the algorithms might be tricky.

C.3.1.7 SSM Parameter Estimation

The SSM uses instantaneous rate equations and provides estimates of catchability coefficients (q) for each fishery. However, up to this point we do not have a formal description of a SSM that includes multiple fisheries operating within the same region at the same time. The two fisheries modeled in the prototype SSM (Canadian Troll, US Troll) operate simultaneously, but in different regions. I think I know what that formulation will look like, but we need to formalize it. Once the model is formulated, we need to answer the following questions:

Q1. If the SSM is fit to data for individual stocks, should the forward simulation model use separate q's for each fishery and stock? The alternative is to fit the SSM to multiple stocks assuming a common q. Is this biologically appropriate (i.e., are all stocks equally vulnerable to a fishing gear)? Is this feasible? How will it be done?

Q2. Regardless of how Q1 is resolved, the q estimates for each fishery will reflect all regulations associated with each fishery during the time frame when the data were collected (e.g., size limits, bag limits, selective fishery rules). If we desire to simulate changes in size limits, bag limits, and selective fishery rules, how will the SSM be modified to reflect these changes? The Lawson and Sampson (1996) model might be appropriate.

Q3. In forward simulation, how will the SSM handle incidental mortalities? The q's estimated by the SSM reflect legal catches and will not provide information about incidental mortalities related to fishing. I believe all incidental mortalities will be absorbed by the instantaneous natural mortality parameter (M) in the SSM, provided it is not assumed to be zero. I suppose we can use auxiliary data to partition M into all types of non-legal catch mortalities. How will we do this?

Q4. The q's estimated by the SSM will be instantaneous rates based on a daily timestep. If a forward simulation model is configured to operate on a weekly or monthly basis, must the model use instantaneous rate equations, or can we convert the q's (and associated effort levels) to what Ricker calls "conditional rates" (i.e., the fraction of a cohort that dies within a given time period) and use other equations?

These questions were discussed at the July 30, 1998 meeting. There were no minutes published for the July 30, 1998 meeting. However, a summary of the July 30, 1998 meeting was included in the minutes for the august 27, 1998 meeting. Below are the comments related to the above four questions are from email correspondence from Jim Norris to Troy Frever.

Q1. Using a common catchability coefficient for each stock is biologically acceptable, desirable, and feasible. Ken thinks he can solve for more than one stock at the same time.

Q2. Any other adjustments to fishing mortalities (e.g., due to bag limits, size limits, drop-offs, etc) can be incorporated into the instantaneous catch equations used by the SSM. However, they cannot be estimated by the SSM ... they must come from outside the SSM (e.g., from the accepted values used by other models, such as the Lawson and Sampson equations used in PM model). The key issue for our coding is that we will have more parameters to deal with, but we knew we would have to deal with them anyway. In many cases these will be fixed parameters that will be constant across stocks, fisheries (or gear types), regions, and timesteps (e.g., drop off rates, mark recognition rates, shaker mortality rates). The bigger issue is for parameter estimation. If those additional parameters will be used with instantaneous rate equations, then they must be input as parameters into the equations during the estimation process. This means that Ken will need more than just catches and efforts to do his estimation. Someone (probably Jim Scott and/or Robert Kope) will have to get these parameters together for Ken before he can go into production mode for

parameter estimation. Jim Scott reported that he still doesn't have all the effort data yet, so this parameter estimation process is getting even more complicated and time consuming.

Q3. Incidental mortalities will have to be handled like the other mortality adjustments discussed above in Q2. Again, these will have to be provided to Ken before he can do the estimations.

Q4. No problem converting q values estimated using instantaneous rate equations in the SSM to a discrete model case. Jim Scott gave me a paper to read about the meaning of "catchability coefficient." This paper deals with exactly the types of questions you were asking me about the definition of q at our last meeting. I'm about half way through and will give you an update next week.

C.3.1.8 Final Thought

It seems that the theoretical model we seek is some combination of the detailed migration sub-model of the SSM with the detailed harvesting sub-model proposed by Lawson and Sampson (1996). The question is: If we include the detailed harvesting sub-model into the SSM, can the SSM estimate all the parameters? Can we make some simplifying assumptions and use auxiliary data to make the problem tractable?

C.3.2 Shaker Algorithm

The following discussion is taken from the minutes of the April 19, 1999 meeting.

The PSC Chinook Model shaker algorithm was difficult to implement in the Coast Model because it relied upon a subjective concept (preterminal vs terminal; ocean net fishery) of which stocks were considered vulnerable to each fishery during a timestep. The new Coast Model shaker algorithm resolves this dilemma by including a “vulnerability” table in each shaker object. Since shakers are computed for each fishery independently, each fishery/region/timestep has a shaker object. There are essentially four different cases (described in the following sections).

C.3.2.1 Non-ocean net fisheries with no terminal stocks

For these fisheries, all cohorts (ages) from all stocks are vulnerable only during the preterminal timestep and region. In the 1998 mode configuration, all troll fisheries, most sport fisheries, and a few net fisheries fall into this category. We call this the “SimpleDrop” shaker method because it does not require a vulnerability table. The input data file looks like this:

```
Fishery 1    # Alaska T
  Method  SimpleDrop
    SubLegalReleaseMortRate  0.255
    DropOffRate              0.008
  end Method
end Fishery
```

Fisheries that have different rates in different years look like this:

```
Fishery 5    # WA/OR T
 Years 1979:1984
  Method  SimpleDrop
    SubLegalReleaseMortRate  0.255
    DropOffRate              0.017
  end Method
 end Years
 Years 1985:1999
  Method  SimpleDrop
    SubLegalReleaseMortRate  0.220
    DropOffRate              0.025
  end Method
 end Years
end Fishery
```

C.3.2.2 Non-ocean net fisheries with at least one terminal stock

For these fisheries, all ages for all non-terminal stocks are vulnerable during the preterminal timestep and region, and all ages for all terminal stocks are vulnerable during the terminal timestep and region. We call this (and cases 3 and 4) the "CustomDrop" method, because it requires a vulnerability table. Here's a sample:

```
Fishery 25    # Col R S
  Method CustomDrop
    SubLegalReleaseMortRate 0.123
    DropOffRate             0.069
    VulnerabilityTable StockXageXtime
      TimeStep 1  # Preterminal
        Ages 2:5  # non-terminal stocks
          Stock 1 Vulnerable # Alaska South SE
          Stock 2 Vulnerable # North/Centr
          Stock 3 Vulnerable # Fraser Early
          Stock 4 Vulnerable # Fraser Late
          Stock 7 Vulnerable # Georgia St. Upper
          Stock 8 Vulnerable # Georgia St. Lwr Nat
          Stock 9 Vulnerable # Georgia St. Lwr Hat
          Stock 10 Vulnerable # Nooksack Fall
          Stock 11 Vulnerable # Pgt Sd Fing
          Stock 12 Vulnerable # Pgt Sd NatF
          Stock 13 Vulnerable # Pgt Sd Year
          Stock 14 Vulnerable # Nooksack Spring
          Stock 15 Vulnerable # Skagit Wild
          Stock 16 Vulnerable # Stillaguamish Wild
          Stock 17 Vulnerable # Snohomish Wild
          Stock 18 Vulnerable # WA Coastal Hat
          Stock 28 Vulnerable # WA Coastal Wild
        end Ages
      end TimeStep
      TimeStep 2  # Terminal
        Ages 2:5  # terminal stocks
          Stock 5 Vulnerable # WCVI Hatchery
          Stock 6 Vulnerable # WCVI Natural
          Stock 19 Vulnerable # UpRiver Brights
          Stock 20 Vulnerable # Spring Creek Hat
```

```
          Stock 21 Vulnerable # Lwr Bonneville Hat
          Stock 22 Vulnerable # Fall Cowlitz Hat
          Stock 23 Vulnerable # Lewis R Wild
          Stock 24 Vulnerable # Willamette R
          Stock 25 Vulnerable # Spr Cowlitz Hat
          Stock 26 Vulnerable # Col R Summer
          Stock 27 Vulnerable # Oregon Coast
          Stock 29 Vulnerable # Lyons Ferry
          Stock 30 Vulnerable # Mid Col R Brights
        end Ages
      end TimeStep
    end VulnerabilityTable
  end Method
end Fishery
```

C.3.2.3 Ocean net fisheries with no terminal stocks

For these fisheries all stocks have ages 2 and 3 vulnerable during the preterminal timestep and ages 4 and 5 vulnerable during the terminal timestep. For example:

```
Fishery 7    # Alaska N
  Method  CustomDrop
    SubLegalReleaseMortRate  0.9
    DropOffRate              0
    VulnerabilityTable StockXageXtime
      TimeStep 1  # Preterminal
        Ages 2:3
          Vulnerable
      end TimeStep
      TimeStep 2  # Terminal
        Ages 4:5
          Vulnerable
      end TimeStep
    end VulnerabilityTable
  end Method
end Fishery
```

C.3.2.4 Ocean net fisheries with at least one terminal stock

These fisheries have ages 2 and 3 from non-terminal stocks vulnerable during the preterminal timestep, ages 4 and 5 from non-terminal stocks vulnerable during the terminal timestep, and all ages from terminal stocks vulnerable during the terminal timestep.

```
Fishery 12    # PgtNth N
  Method  CustomDrop
    SubLegalReleaseMortRate  0.9
    DropOffRate              0
    VulnerabilityTable StockXageXtime
      TimeStep 1  # Preterminal
        Ages 2:3  # non-terminal stocks
          Stock 1 Vulnerable # Alaska South SE
          Stock 2 Vulnerable # North/Centr
          Stock 3 Vulnerable # Fraser Early
          Stock 4 Vulnerable # Fraser Late
          Stock 5 Vulnerable # WCVI Hatchery
          Stock 6 Vulnerable # WCVI Natural
```

```
    Stock 7 Vulnerable # Georgia St. Upper
    Stock 8 Vulnerable # Georgia St. Lwr Nat
    Stock 9 Vulnerable # Georgia St. Lwr Hat
    Stock 18 Vulnerable # WA Coastal Hat
    Stock 19 Vulnerable # UpRiver Brights
    Stock 20 Vulnerable # Spring Creek Hat
    Stock 21 Vulnerable # Lwr Bonneville Hat
    Stock 22 Vulnerable # Fall Cowlitz Hat
    Stock 23 Vulnerable # Lewis R Wild
    Stock 24 Vulnerable # Willamette R
    Stock 25 Vulnerable # Spr Cowlitz Hat
    Stock 26 Vulnerable # Col R Summer
    Stock 27 Vulnerable # Oregon Coast
    Stock 28 Vulnerable # WA Coastal Wild
    Stock 29 Vulnerable # Lyons Ferry
    Stock 30 Vulnerable # Mid Col R Brights
  end Ages
end TimeStep
TimeStep 2  # Terminal
  Ages 4:5  # non-terminal stocks
    Stock 1 Vulnerable # Alaska South SE
    Stock 2 Vulnerable # North/Centr
    Stock 3 Vulnerable # Fraser Early
    Stock 4 Vulnerable # Fraser Late
    Stock 5 Vulnerable # WCVI Hatchery
    Stock 6 Vulnerable # WCVI Natural
    Stock 7 Vulnerable # Georgia St. Upper
    Stock 8 Vulnerable # Georgia St. Lwr Nat
    Stock 9 Vulnerable # Georgia St. Lwr Hat
    Stock 18 Vulnerable # WA Coastal Hat
    Stock 19 Vulnerable # UpRiver Brights
    Stock 20 Vulnerable # Spring Creek Hat
    Stock 21 Vulnerable # Lwr Bonneville Hat
    Stock 22 Vulnerable # Fall Cowlitz Hat
    Stock 23 Vulnerable # Lewis R Wild
    Stock 24 Vulnerable # Willamette R
    Stock 25 Vulnerable # Spr Cowlitz Hat
    Stock 26 Vulnerable # Col R Summer
    Stock 27 Vulnerable # Oregon Coast
    Stock 28 Vulnerable # WA Coastal Wild
    Stock 29 Vulnerable # Lyons Ferry
    Stock 30 Vulnerable # Mid Col R Brights
  end Ages
  Ages 2:5  # terminal stocks
    Stock 10 Vulnerable # Nooksack Fall
    Stock 11 Vulnerable # Pgt Sd Fing
    Stock 12 Vulnerable # Pgt Sd NatF
    Stock 13 Vulnerable # Pgt Sd Year
    Stock 14 Vulnerable # Nooksack Spring
    Stock 15 Vulnerable # Skagit Wild
    Stock 16 Vulnerable # Stillaguamish Wild
    Stock 17 Vulnerable # Snohomish Wild
  end Ages
```

```
      end TimeStep
    end VulnerabilityTable
  end Method
end Fishery
```

C.3.3 Results of multi-phase catch ceiling algorithm test.

The following discussion is taken from the minutes of the January 19, 1999 meeting. Note that although this discussion states that the Coast Model does not have an iteration capability to handle catch ceilings over multiple timesteps, such an algorithm was developed shortly thereafter. We include this discussion here because it is informative about how catch ceiling management can behave.

Some fisheries in the PSC Chinook Model that are controlled by catch ceilings have harvests in both the preterminal and terminal time steps (we call these multi-phase ceiling fisheries). The algorithm used to compute catches for these fisheries assumes (1) a single catch ceiling that covers both time steps, and (2) that the input fishing effort levels in both time steps are adjusted by the same relative amounts in order to meet the catch ceiling. There is no analytical solution for catches in multi-phase ceiling fisheries, because any change in the preterminal effort changes the stock abundances vulnerable to terminal effort. Instead, the algorithm iterates over both time steps, and on each iteration scales the effort levels in the preterminal and terminal timesteps by the same relative amount until the total catch from both timesteps equals the desired catch ceiling.

The NMFS Coast Model allows catch ceilings to be specified for individual timesteps and regions, but does not have an algorithm to ensure equal relative effort levels across timesteps. Thus, the NMFS Coast Model algorithm works fine for ceilinged fisheries that have catches only in one timestep/region (we call these single-phase ceiling fisheries), but does not work for multi-phase ceiling fisheries (i.e., does not always give the same catches as the PSC Chinook Model).

We were interested in knowing the extent of the differences between the two algorithms. This report describes a simple, but informative, experiment to gain insight into the differences.

We prepared a *.cei file that included 16 fisheries, four of which were multi-phase ceiling fisheries (Alaska Net, Northern Net, Central Net, and WCVI Sport). The base period was defined to be 1979-1984. From 1985-1994, ceilings were forced (i.e., model catches were required to equal the ceiling exactly) for all but two fisheries (North and South Puget Sound Sport), which were forced from 1985-1993. Catch ceilings during the period 1995-1997 were unforced and set extremely high to simulate fishery policy (FP) control (i.e., model catches were always below the ceiling and were not adjusted upward to equal the ceiling; thus, the ceilings had no effect). For 1998 and beyond, ceilings were unforced and set to the average 1991-1994 catches.

Both the PSC Chinook Model and the NMFS Coast Model were run using the same *.cei file data. A catch file (*cat.prn) and escapement file (*esc.prn) were printed for each model. The absolute values of the catch and escapement differences between models were computed.

Table 5 and 0 give the percentage change from the PSC Chinook Model to the NMFS Coast Model for catches and escapements. The catch results are summarized below:

- Both models gave the same results for the base period 1979-1984;
- Catches in all ceilinged fisheries were the same when the ceilings were forced;
- Catches in all ceilinged fisheries were not the same when ceilings were unforced and set very high (1995-1997);
- Beyond 1997, catches in some ceilinged fisheries were the same, but in others were off by small amounts (the primary exception was the WCVI Sport fishery which had catches up to 10% off);
- Catches in all non-ceilinged fisheries were not the same for all years beyond the base period.

The escapement results are summarized below:

- Escapements were different for all years beyond the base period;
- The greatest differences were in the WCVI stocks (RBT and RBH), which were up to 14% off during the period 1985-1993;
- The GSQ stock also was off by up to 12% during the period 1985-1994.

When ceilings are forced, the PSC Chinook Model algorithm maintains a constant relative EFFORT level between the preterminal and terminal timesteps, whereas the NMFS Coast Model algorithm maintains a constant relative CATCH level between the preterminal and terminal timesteps. Thus, as long as catch ceilings are forced, both algorithms give the same total catch, even for the multi-phase ceiling fisheries (e.g., 1985-1994 period in Table 5). However, each algorithm will distribute the total catch differently between the preterminal and terminal timesteps for the multi-phase ceiling fisheries due to the different assumptions. The difference in catch distribution in the multi-phase ceiling fisheries affects the relative abundances of the stocks, leading to differences in catches in non-ceilinged fisheries and to differences in spawning escapements for individual stocks. The abundance and escapement differences are most pronounced for stocks that are heavily harvested by a multi-phase ceiling fishery, such as the RBH and RBT stocks in the WCVI Sport fishery.

When ceilings are not forced, the effects on catches can be variable, depending on the level of the ceiling. If the ceiling is very low such that the unconstrained model catch in a multi-phase ceiling fishery is always above both the associated preterminal and terminal ceilings in the NMFS Coast Model, the net effect is the same as when ceilings are forced because the catches always have to be reduced to meet the ceiling. For example, during the period 1998-2005 the catches for fisheries 1-3, 5-9, and 21 were identical for both models. For the other ceilinged fisheries, the catches were generally different by a small amount (less than one percent), except for the WCVI Sport fishery, which had differences up to 10%.

The WCVI Sport fishery is an interesting case that deserves further discussion. During the period of forced ceilings (1985-1994), the total catch in the WCVI Sport fishery is the same under both algorithms, but the distribution of the catch is probably very different. This large difference in distribution leads to the large differences in escapements for the RBH and RBT stocks during this period. Once the ceilings become unforced, the catch differences between the two algorithms are large due to the large differences in stock abundances. One unexplained anomaly is that the catch differences are very small during the first three years (1995-1997) of the unforced ceilings.

Table 5 Percent change in ceilinged fishery catches by year between PSC and NMFS catch ceiling algorithms. The four multi-phase ceiling fisheries (7, 8, 9, 20) are listed at the far right side of the table.

Year	1	2	3	4	5	6	18	19	21	22	23	24	7	8	9	20
1979																
1980																
1981																
1982																
1983																
1984																
1985																
1986																
1987																
1988																
1989																
1990																
1991																
1992																
1993																
1994										0.4	0.3					
1995	0.0	0.1	0.1	0.4	0.4	0.6	0.2	0.0	0.4	0.3	0.3	0.5	0.2	0.2	0.2	0.5
1996	0.2	0.2	0.2	0.2	0.4	0.4	0.4	0.1	0.3	0.1	0.2	0.3	0.4	0.4	0.4	0.3
1997	0.4	0.3	0.3	0.1	0.2	0.3	0.5	0.0	0.2	0.1	0.2	0.2	0.5	0.4	0.2	0.2
1998				0.1			0.2	0.0		0.1	0.2					
1999				0.2				0.0		0.2	0.2					
2000				0.3			0.1	0.0		0.2	0.2	0.3				0.3
2001				0.2			0.0	0.0		0.2	0.1					
2002				0.2			0.1	0.1		0.2	0.2					
2003				0.2			0.2	0.1		0.2	0.2					
2004				0.2			0.1	0.1		0.2	0.1					
2005				0.2			0.1	0.1		0.1	0.1					
Min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Avg	0.0	0.0	0.0	0.1	0.0	0.0	0.1	0.0	0.0	0.1	0.1	0.0	0.0	0.0	0.0	0.0
Max	0.4	0.3	0.3	0.4	0.4	0.6	0.5	0.1	0.4	0.4	0.3	0.5	0.5	0.4	0.4	0.5

Table 6 Percent change in non-ceilinged fishery catches by year between PSC and NMFS catch ceiling algorithms.

Year	10	11	12	13	14	15	16	17	25
1979									
1980									
1981									
1982									
1983									
1984									
1985	0.1		0.0	0.0	0.0	0.1	0.0	0.2	0.0
1986	0.3	0.2	0.1	0.1	0.2	0.3	0.2	0.2	0.1
1987	0.4	0.1	0.1	0.2	0.4	0.3	0.2	0.3	0.1
1988	0.2	0.2	0.1	0.1	0.2	0.2	0.3	0.1	0.3
1989	0.2	0.2	0.1	0.2	0.1	0.2	0.1	0.5	0.2
1990	0.4	0.1	0.2	0.5	0.3	0.4	0.4	0.4	0.1
1991	0.8	0.0	0.4	0.9	0.7	0.7	0.1	0.5	0.0
1992	0.9	0.1	0.4	0.9	0.9	0.8	0.1	0.6	0.0
1993	0.8	0.2	0.5	0.8	1.0	0.7	0.2	0.9	0.0
1994	0.6	0.2	0.5	0.7	0.9	0.4	0.2	0.8	0.1
1995	0.1	0.2	0.4	0.3	0.6	0.3	0.1	0.7	0.1
1996	0.6	0.1	0.2	0.2	0.2	0.2	0.3	0.5	0.2
1997	0.6	0.2	0.1	0.1	0.1	0.0	0.2	0.5	0.1
1998	0.4	0.1	0.2	0.2	0.0	0.1	0.2	0.5	0.1
1999	0.0	0.2	0.3	0.3	0.1	0.2	0.1	0.5	0.1
2000	0.1	0.2	0.2	0.3	0.2	0.3	0.0	0.4	0.0
2001	0.0	0.1	0.2	0.3	0.3	0.3	0.0	0.4	0.1
2002	0.1	0.1	0.2	0.3	0.2	0.3	0.0	0.4	0.1
2003	0.2	0.1	0.2	0.3	0.2	0.3	0.1	0.4	0.1
2004	0.1	0.1	0.2	0.3	0.2	0.3	0.1	0.4	0.1
2005	0.2	0.1	0.2	0.3	0.2	0.3	0.0	0.3	0.1
Min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Avg	0.3	0.1	0.2	0.3	0.3	0.2	0.1	0.4	0.1
Max	0.9	0.2	0.5	0.9	1.0	0.8	0.4	0.9	0.3

Table 7 Percent change in stock escapements by year between PSC and NMFS catch ceiling algorithms (fisheries 1-15).

Year	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1979															
1980															
1981															
1982															
1983															
1984															
1985	0.5	2.1	0.4	0.0	7.4	4.8	7.6	1.7	1.2	0.2	0.0	0.0	0.3	0.0	0.1
1986	1.1	3.6	0.7	0.0	12.7	7.9	11.1	2.2	2.9	0.5	0.0	0.0	0.5	0.1	0.3
1987	0.3	0.7	0.5	0.1	0.7	0.7	1.1	0.4	0.2	0.0	0.1	0.1	0.1	0.2	0.5
1988	0.8	1.0	0.6	0.1	7.7	5.9	5.8	1.4	1.5	0.2	0.1	0.0	0.2	0.1	0.4
1989	1.4	1.6	0.7	0.1	6.2	10.5	3.6	2.1	3.7	0.1	0.2	0.1	0.3	0.1	0.4
1990	0.9	2.8	0.6	0.6	9.4	7.7	12.2	2.7	4.1	0.2	0.4	0.4	0.4	0.2	0.4
1991	0.6	0.1	0.7	1.0	13.6	12.3	7.6	0.7	1.1	1.1	0.9	0.8	0.7	0.3	1.0
1992	1.0	1.6	0.9	1.4	9.3	9.6	6.6	1.7	1.2	1.2	0.7	0.7	0.7	0.3	1.2
1993	1.0	0.2	1.0	1.2	12.5	13.6	3.9	2.0	1.3	0.9	0.7	0.7	0.5	0.4	1.0
1994	0.4	1.0	0.7	1.0	2.3	6.7	6.1	0.7	0.6	0.6	0.5	0.4	0.4	0.4	1.0
1995	0.1	1.1	0.5	0.9	0.4	2.5	3.7	0.2	0.1	0.3	0.2	0.2	0.7	0.4	1.0
1996	0.5	0.1	0.5	0.9	0.0	3.2	3.1	0.8	0.2	0.1	0.0	0.2	0.7	0.3	0.9
1997	0.5	0.1	0.3	0.7	0.0	3.1	3.4	0.7	0.7	0.0	0.0	0.1	0.5	0.3	0.7
1998	0.7	0.7	0.3	0.6	5.5	7.3	6.8	0.9	1.2	0.2	0.1	0.2	0.2	0.3	0.8
1999	0.5	0.4	0.4	0.7	5.2	6.2	7.3	0.7	0.7	0.0	0.2	0.2	0.6	0.5	1.0
2000	0.5	0.4	0.4	0.8	5.1	6.2	5.0	0.7	0.8	0.0	0.2	0.1	0.6	0.3	0.9
2001	0.5	0.4	0.4	0.7	5.1	6.9	4.5	0.7	0.6	0.0	0.2	0.1	0.4	0.4	0.9
2002	0.6	0.5	0.4	0.6	5.1	7.4	5.6	0.6	0.4	0.0	0.2	0.1	0.3	0.5	0.9
2003	0.6	0.5	0.4	0.6	5.0	7.3	7.5	0.5	0.4	0.0	0.2	0.2	0.5	0.4	1.0
2004	0.5	0.5	0.4	0.6	5.1	7.3	7.0	0.5	0.3	0.0	0.2	0.2	0.6	0.4	0.9
2005	0.6	0.5	0.4	0.6	5.1	7.5	5.5	0.5	0.4	0.0	0.2	0.2	0.4	0.4	0.9
Min	0.1	0.1	0.3	0.0	0.0	0.7	1.1	0.2	0.1	0.0	0.0	0.0	0.1	0.0	0.1
Avg	0.6	0.9	0.5	0.6	5.9	6.9	6.0	1.1	1.1	0.3	0.2	0.2	0.5	0.3	0.8
Max	1.4	3.6	1.0	1.4	13.6	13.6	12.2	2.7	4.1	1.2	0.9	0.8	0.7	0.5	1.2

Table 7 Continued (fisheries 16-20).

Year	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1979															
1980															
1981															
1982															
1983															
1984															
1985	0.2	0.1	1.2	0.6	0.0	0.1	0.0	0.6	0.0	0.0	0.1	2.7	3.2	0.0	0.5
1986	0.6	0.3	0.6	1.1	0.1	0.0	0.0	0.9	0.2	0.1	0.4	0.4	0.0	0.0	0.0
1987	0.8	0.5	0.3	0.1	0.0	0.1	0.1	0.0	0.2	0.1	0.1	0.0	0.2	0.0	0.3
1988	0.7	0.4	1.4	0.9	0.0	0.1	0.1	0.5	0.2	0.1	0.1	1.4	1.9	0.3	0.3
1989	0.9	0.4	1.3	0.1	0.1	0.3	0.4	0.3	0.2	0.1	0.0	0.3	0.7	0.0	0.1
1990	2.5	0.5	0.2	0.1	0.3	1.1	1.1	0.3	0.2	0.1	0.3	0.3	2.3	0.0	0.9
1991	3.3	0.9	0.0	0.2	0.6	2.6	1.8	0.3	0.2	0.2	0.1	0.1	0.4	1.8	0.1
1992	4.8	1.1	1.6	0.3	0.9	2.8	1.7	0.9	0.3	0.2	0.2	0.3	0.4	1.7	0.4
1993	4.7	1.1	0.5	0.1	0.8	2.2	1.8	0.3	0.3	0.2	0.1	0.2	2.2	1.3	0.2
1994	4.7	1.0	0.6	0.5	0.4	1.2	1.9	0.2	0.3	0.2	0.1	0.2	0.7	1.3	0.1
1995	3.7	0.9	0.5	0.3	0.3	0.6	1.8	0.1	0.2	0.1	0.1	0.2	0.5	1.6	0.2
1996	3.6	0.5	0.2	0.2	0.6	0.1	1.7	0.1	0.0	0.0	0.1	0.0	0.2	1.6	0.3
1997	3.5	0.4	0.0	0.1	0.2	0.0	1.8	0.1	0.0	0.0	0.1	0.0	0.2	1.3	0.2
1998	3.6	0.6	0.7	0.5	0.1	0.2	2.0	0.4	0.0	0.0	0.1	0.4	0.7	1.6	1.2
1999	4.1	0.7	0.6	0.4	0.1	0.4	2.0	0.3	0.1	0.1	0.0	0.3	0.3	1.6	0.8
2000	4.1	0.6	0.5	0.3	0.2	0.4	2.0	0.3	0.1	0.1	0.0	0.2	0.3	1.7	0.6
2001	4.3	0.6	0.4	0.2	0.2	0.4	2.1	0.3	0.1	0.1	0.1	0.2	0.3	1.7	0.6
2002	4.3	0.7	0.5	0.3	0.1	0.4	2.2	0.3	0.1	0.1	0.1	0.3	0.3	1.9	0.6
2003	4.7	0.7	0.5	0.3	0.2	0.4	2.3	0.3	0.1	0.1	0.1	0.3	0.3	1.8	0.6
2004	5.0	0.7	0.4	0.3	0.1	0.4	2.3	0.3	0.1	0.1	0.0	0.3	0.2	1.9	0.6
2005	5.3	0.7	0.4	0.3	0.2	0.4	2.3	0.3	0.1	0.1	0.0	0.3	0.3	1.9	0.6
Min	0.2	0.1	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Avg	3.3	0.6	0.6	0.3	0.3	0.7	1.5	0.3	0.1	0.1	0.1	0.4	0.7	1.2	0.4
Max	5.3	1.1	1.6	1.1	0.9	2.8	2.3	0.9	0.3	0.2	0.4	2.7	3.2	1.9	1.2

C.4 Maturation Process

Initially, we envisioned using the migration process to simulate maturation. At the November 6, 1997 meeting the model committee pointed out that this formulation would not work. Jim Norris posted the following discussion on the web site on December 10, 1997.

First, I want to thank members of the Model Committee for pointing out our failure to properly code the maturation process. This was a significant oversight on our part. Fortunately, at this stage in development it is fairly easily fixed. Below is a summary of our proposed changes.

C.4.1 The Biological Process

Our understanding is that for almost all salmon stocks, individuals within a cohort may or may not mature and return in a given year. At the individual fish level, the decision to return is determined by both genetic factors and environmental factors. On the genetic side, each salmon species has a fairly consistent maturation process (e.g., chinook mature over several years; coho tend to mature all in same year). On the environmental side, individual fitness (e.g., general body chemistry) during the spring or early summer may determine which individuals actually decide to migrate home, and when.

C.4.2 Mathematical Modeling Problem

The above biological realities imply that at any given time and region a cohort may be divided into two distinct components (immature and mature) that will have different migration patterns. The modeling question is whether or not it is necessary to model each component of the cohort separately.

If we assume no fishing mortality, ignore mature and immature status, divide time and space into discrete units, and track the movement of each individual fish, at any time step we could determine the fraction of the fish that move from each cell to each other cell. From these data we could define transition matrices for each time step that fully describe the movement of the entire cohort, regardless of maturation status. In this perspective (no fishing), it does not seem necessary to track mature and immature fish separately.

However, the fishing process will alter the relative composition of immature/mature fish within a time/area cell. Since these two components have different migration patterns, the migration pattern for the combined cohort (immature and mature) can no longer be expressed by a single transition matrix.

C.4.3 New Main Engine

We will add a new maturation process to the main engine just before the migration process. Thus, the main loop in the computation engine will look something like this:

```
YearInit(year);

   TimeStepIter time(clock);
   while (++time) {
    naturalMortalityManager.takeNaturalMortality(clock);
    FisheryManager.takeFishingMortality(clock);
    spawningManager.spawnCohorts(clock);
 NEW ==> maturationManager.maturateCohorts(clock);
    migrationManager.migrateCohorts(clock);
   }

   for (i = 0; i < Stocks.num(); ++i) {
    Stocks[i].year_wrapup();
   }
```

It may seem odd to place the maturation process after the spawning process. We do this because for modeling purposes the maturation process is most closely linked to the migration process rather than the spawning process. I don't think the order matters much because the spawning process will only be activated in certain time steps and/or regions. We are still evaluating this ordering and haven't made a firm decision yet. Any comments?

C.4.4 Cohort Objects and Data Tracking

At the meeting we discussed two possible approaches to coding the maturation process. One was to treat the immature and mature components as separate cohorts; the other was to keep separate abundance vectors for immature and mature components within the same cohort.

We note that maturation is the only model process that transfers fish from one abundance vector (immature) to another (mature) within the same time/area cell. We also note that the immature and mature components share many characteristics (e.g., species, stock, brood year, age, mark status, tag status). Despite these considerations that suggest keeping immature and mature vectors within the same cohort, we decided to use the first alternative and treat immature and mature fish as separate cohorts. Our reasons are:

1. From a biological perspective, the immature and mature components are biologically separate cohorts, in the sense that they have significantly different demographic characteristics, mainly a different migration pattern. They also may have a different growth rate or size distribution.
2. From a coding perspective, tracking two components within the same cohort would be messy (separate abundance vectors, separate transition matrices, maybe separate growth functions, etc.) and would violate our basic concept of what a cohort is.

At config time, all the cohorts can still be created. Mature/Immature will be a part of the CohortID. The abundance vectors for mature cohorts will be initialized at zero, and will be filled in during the maturation process at appropriate time steps.

This approach will require twice the number of cohorts as previously planned, and twice the data storage requirements (assuming we need to track data for immature and mature cohorts separately).

C.4.5 Relationship to State Space Model

An important consideration is how the maturation process will be estimated. As a first step, I've added the maturation process to the SSM. I have no idea whether or not the parameters in the new formulation can be estimated, but the exercise helps clarify some of the modeling questions. I'm hoping that these ideas will spark further ideas from those more familiar with the estimation procedures.

Recall that the SSM formulation (in matrix notation) is:

$$n(t) = M(t)\, S(t)\, n(t-1)$$

$$c(t) = H(t)\, n(t)$$

This formulation assumes that the cohort abundance being tracked is the mature cohort. Or stated another way, this model assumes that the maturation process occurred prior to time $t = 0$. This may be a reasonable assumption for coho, but not for chinook.

Now consider tracking the mature and immature components separately. Let

$n'(t)$ = immature abundance vector at time t;

$n''(t)$ = mature abundance vector at time t;

$M'(t)$ = transition matrix for immature fish;

$M''(t)$ = transition matrix for mature fish;

$m(t)$ = maturation matrix at time t,

where $m(t)$ is a diagonal matrix with diagonal elements equal to the maturation rates by region for time step t and all other elements = 0. In most cases I suspect that at any time t, the elements of $m(t)$ will be assumed identical, implying that the maturation process is the same over all regions.

To simplify, assume that immature fish do not migrate during the modeling period, and thus M'(t) = I (the identity matrix). If we further assume that mature and immature fish have the same survival and harvest matrices (in practical terms this probably means assuming they have the same size, vulnerability to gear, feeding behavior, etc), we can write the SSM as:

$$n'(t) = [I - m(t)]\, S(t)\, n'(t-1)$$

$$n''(t) = M''(t)\, [S(t)\, n''(t) + m(t)\, S(t)\, n'(t)]$$

$$c(t) = H(t)\, [n'(t) + n''(t)]$$

In considering how to model the maturation process, I see three key parameters: (1) what is the total maturation rate for the given year--15% ? 75% ?; (2) when is the peak maturation date--the date on which the most immature fish become mature fish--Julian day 156?, 275?; and (3) over what time range does the maturation process continue--two weeks?, two months?.

If one assumes that the maturation process is independent of region (i.e., for any given time t, the diagonal elements of m(t) are identical) and fixes one or two of the parameters mentioned above (based on other biological information), it seems that the maturation process could be modeled with only one or two additional parameters to estimate.

C.5 Production Processes

C.5.1 Pre-Spawning Mortality And Production Functions

The following discussion is from the minutes of the October 6, 1998 meeting.

The PSC Chinook Model uses a truncated Ricker function for natural stocks. The Ricker A parameter and an optimum escapement level are the input parameters. The Ricker B parameter is determined from the Ricker A parameter using Hilborn's Approximation. If there is no additional mortality after all harvesting (this is true for all but 3 stocks), the maximum escapement level at which to truncate the function is given by RickerB/RickerA.

For three Columbia River stocks, the escapement from the fisheries is adjusted by a pre-spawning survival factor (called the IDL, or "inter-dam loss", factor). For these stocks, the maximum escapement level is increased to account for pre-spawning mortality. Here's the code.

```
FUNCTION FRicker (ESC, Stk%, Yr%)
  A = RickerA(Stk%)
  B = Optimum(Stk%) / (.5 - .07 * A)
  FRicker = ESC * EXP(A * (1 - ESC / B))
  '..... If escapement exceeds level producing maximum recruitment,
  '..... keep recruits at maximum.  cf Ricker 1975, p. 347, eq. 10
  MaxEsc! = B / (A * IDL!(Stk%, Yr%))
  IF ESC > MaxEsc! THEN
    FRicker = B * EXP(A - 1) / A
  END IF
END Function
```

There was a lot of discussion about this algorithm. The main point of contention was why the maximum escapement value should be adjusted if the escapement value ("ESC") being passed into the function already had been adjusted for pre-spawning escapement. My understanding (readers please correct me if I'm wrong!) is the following.

The Ricker function parameters are estimated from observed spawners and observed recruitment to the fisheries. Thus, the parameters do not incorporate any non-fishing mortality after fishing but before spawning. Since the parameters do not incorporate pre-spawning survival, the computed maximum

escapement doesn't incorporate it either. That is, the maximum escapement parameter assumes there is no pre-spawning mortality. Thus, adjusting the maximum escapement level by pre-spawning mortality is to correct the maximum escapement parameter value, not to correct the escapement passed into the function.

In the new model we treat the pre-spawning mortality as a form of non-fishing, or natural, mortality that may occur during any timestep and in any region, depending on how the user configures the natural mortality process. Thus, the new model may be configured to have pre-spawning mortality during several timesteps and regions, depending on the model resolution. For example, separate "natural" mortalities could be assessed as fish pass each dam.

To extend the concept of pre-spawning mortality to variable timesteps and regions, we need a parameter that represents natural mortality of the escapement (all age classes) from the fisheries to the spawning grounds. Currently, the stock/year specific IDL parameters serve this purpose. An analogous parameter for the generalized case is the weighted average of the IDLs for each age class in the escapement, where the weights are the relative contributions of each age class to the total escapement. The current algorithm is a special case of the weighted average algorithm. Troy reported that he has already implemented this general structure and it validates with the old code.

C.5.2 Variable Truncated Ricker Function

The following is from an email message from Jim Norris to the model committee dated March 18, 1999 describing the need to create a variable truncated Ricker production function. This discussion is closely related to that presented above. No response from the model committee was received.

Hello All,

In the process of validating the new Coast Model (the preferred name for the model we have been working on) Troy and I discovered a behavior in the Ricker function of the CTC model that we would like to have clarified. The behavior only occurs for natural stocks that have variable IDLs. In the 1995 configuration we have been using for validation, only the URB stock falls into this category. In the 1998 version, the LYF stock also will be in this category.

In order to explain the problem, let's use the following definitions related to a Ricker function:

ESC = adult (ages 3, 4, 5) escapement arriving on the spawning grounds (i.e., after Inter-Dam Losses have been removed);

OPT = optimum escapement level (i.e., number of spawners that produces the greatest difference between the exact replacement line and recruitment line);

Smax = escapement level that produces the maximum number of recruits (i.e., the highest point on the Ricker function and mathematically = RickerB/RickerA parameters);

MaxEsc = an intermediate variable used by the PSC chinook model function FRicker to adjust Smax for inter-dam losses (= Smax/IDL). Note that since IDL is usually < 1, MaxEsc is usually > Smax.

Here is the FRicker code from the CTC model:

```
' ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»
                 FUNCTION FRicker (ESC, Stk%, Yr%)
' ÉÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ»
' º Purpose:  Computes the production, expressed in terms of  º
' º           adult equivalents using a truncated form of a   º
' º           Ricker stock-recruitment function.  Prior to    º
' º           1988, the model did not allow spawning escmts toº
' º           exceed escapement goals.  However, not all      º
```

```
' º           stocks have terminal fisheries capable of       º
' º           controlling escapements.  While escapements are º
' º           allowed at any level, production is truncated atº
' º           the maximum.  This formulation thus prevents    º
' º           production from decreasing at high levels of    º
' º           spawning escapements.  An optimistic view may   º
' º           result, however, production response at high    º
' º           escapement levels is unknown for nearly all     º
' º           stocks.                                         º
' º                                                           º
' º Arguments: Esc  = Age 3+ spawning escapement              º
' º            Stk% = Stock index number                      º
' º                                                           º
' º Inputs:    IDL!()                                         º
' º            Optimum()                                      º
' º            RickerA()                                      º
' º                                                           º
' º Called By: CatchTermFinal                                 º
' º                                                           º
' º Output:    Adult equivalent production                    º
' º                                                           º
' º Externals: None.                                          º
' º                                                           º
' º Module:    CHSIMUL.DOC                                    º
' º                                                           º
' ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ1/4
A = RickerA(Stk%)
B = Optimum(Stk%) / (.5 - .07 * A)
FRicker = ESC * EXP(A * (1 - ESC / B))

'..... If escapement exceeds level producing maximum recruitment,
'..... keep recruits at maximum.  cf Ricker 1975, p. 347, eq. 10
MaxEsc! = B / (A * IDL!(Stk%, Yr%))
IF ESC > MaxEsc! THEN
  FRicker = B * EXP(A - 1) / A
END IF

END FUNCTION
```

For all natural stocks, prior to passing ESC into FRicker, ESC is truncated to OPT if the SkipMSH variable = 0. For the URB stock, SkipMSH = 1, so ESC is not truncated to OPT. Thus, ESC can be quite large when passed into FRicker.

If ESC < MaxEsc, then ESC is used to compute recruits. Note that it is possible that Smax < ESC < MaxEsc, in which case recruits are computed using the upper portion of the descending right limb of the Ricker function.

However, if ESC > MaxEsc, then the number of recruits is computed using Smax. This is because the variable B*EXP(A-1)/A is equivalent to computing the Ricker function using ESC = Smax = B/A.

The net result is a production function that:

- increases from zero to maximum recruitment as spawners increase from 0 to Smax;
- then descends when spawners are between Smax and MaxEsc:
- then jumps up to the maximum recruitment again whenever spawners > MaxEsc.

For IDL values > 0.8, it doesn't make too much difference. But when the IDL values get smaller, the gap between Smax and MaxEsc gets bigger.

This seems like a strange function. Is this what was intended?

-- Jim

C.5.3 Adult Equivalent Factors In Production Functions

The following discussion is from the minutes of the October 6, 1998 meeting.

The production functions in the PSC Chinook Model relate spawners to ADULT recruitment. Adult recruitment is converted back to "Age1Fish" by a factor called "RectAtAge1". In essence, the RectAtAge1 factor defines the relationship between adult recruitment and Age1Fish in the equilibrium condition with no harvesting. Thus, the RectAtAge1 factor is defined by the maturation and natural mortality schedules for each stock. For the Coast Model, three situations need to be considered.

(1) If the maturation and natural mortality schedules do not change over time and space, each stock has a unique RectAtAge1 factor. About 2/3 of the stocks in the PSC Chinook Model fall in this category. Note that the RectAtAge1 factors could be computed before a model run and passed in as input data along with other production function parameters.

(2) If the schedules do change over time and space, then each stock will have a different RectAtAge1 factor for each year. About 1/3 of the stocks in the PSC Chinook model fall in this category. These stocks have constant natural mortality schedules, but variable maturation schedules. At the start of each year, a new RectAtAge1 factor is computed for each stock. As with case (1), since all schedules are included with the input data, the RectAtAge1 factors could be computed before a model run and passed in as input data along with other production function parameters.

(3) Cases (1) and (2) can be generalized to variable timesteps and regions, provided the timestep and region definitions and the maturation and natural mortality schedules for each stock are all defined during the model configuration. We envision the Coast Model will have timesteps and regions defined during configuration, but we would like to allow for the possibility that maturation and survival schedules may be determined dynamically based on model predicted environmental conditions.

Jim and Troy proposed that for forward simulation runs RectAtAge1 factors be considered as part of the production parameters for each stock and should become part of input data. This should be the case even if natural mortality and maturation schedules are determined dynamically. In other words, the model user must have some previously estimated relationship between adult recruits and Age1Fish for each stock and year. If a model dynamically computes natural mortality and maturation schedules, the model predicted RectAtAge1 factors could be stored and reported as output.

C.6 Migration Process

C.6.1 Synthesizing Commonly Used Migration Algorithms

An important goal of this project was to synthesize commonly used migration algorithms into a common mathematical framework. The following report was presented by Jim Norris at the June 19, 1998 meeting. It concludes that the State Space Model migration matrix formulation can be used to represent all commonly used migration algorithms.

C.6.1.1 Background

The two salmon management models currently used most extensively for fisheries that impact stocks listed under the ESA are the Pacific Salmon Commission (PSC) Chinook Model and the Fishery Regulation Analysis Model (FRAM). Neither of these models include specific migration algorithms to simulate the movement of fish through a gauntlet fishery. Three new models have been proposed to simulate salmon migration more accurately: a State Space Model (Newman, 1995), the PSC Selective Fishery Model (PSC 1995), and the Proportional Migration (PM) Model (Moore et al., 1996). The purpose of this report is to describe the migration algorithms used in these five salmon management models.

The State Space Model is the most general of the five models and has been proposed for use in a new salmon life cycle model. Thus, this report describes the existing algorithms in terms common with the State Space Model.

C.6.1.2 Notation

The notation used in this report follows that commonly used by the State Space Model. I describe the migration algorithms with reference to a single cohort of fish. A cohort may be a single age class from a single stock, or it may be a marked/tagged sub-component of that age class. No stock or age subscripts are used to make the notation easier to read. All of the models use discrete time steps and the notation is defined with respect to time interval [t-1, t). Table 8 lists the notation and Figure 1 illustrates the notation graphically.

Table 8 Common notation used in this report (also see Figure 1).

Variable	Definition
<u>Index Variables</u>	
t	Time index.
r	Geographic region index. For models in which there is a one-to-one correspondence between geographic region and fishery, r also indexes fishery.
R	The total number of geographic regions in the model.
<u>State Variables</u>	
$n_{r,t}$	Abundance of fish in region r at the start of time interval $[t, t+1)$.
$c_{r,t}$	Observed catch in region r during the time interval $[t,t+1)$.
$ns_{r,t-1}$	Abundance of fish in region r that do not suffer natural mortality and/or fishing mortality during the time interval $[t-1,t)$. Equal to the abundance of fish in region r at the end of the time interval $[t-1,t)$ just prior to instantaneous migration.

Variable	Definition
Natural Mortality and Survival Parameters	
$M_{r,t}$	Instantaneous natural mortality rate in region *r* during the time interval [*t*-1,*t*).
$\boldsymbol{u}_{r,t}$	Fraction of the cohort in region *r* killed by natural mortality during the time interval [*t*-1,*t*).
$s_{r,t}$	Total survival rate in region *r* during the time interval [*t*-1,*t*).
Fishing Mortality Parameters	
$E_{r,t-1}$	Amount of fishing effort in region *r* during the time interval [*t*-1,*t*).
$h_{r,t-1}$	Regional harvest rate defined as the fraction of the cohort located in region *r* harvested as legal catch in region *r* during the time interval [*t*-1,*t*).
$F_{r,t-1}$	Instantaneous total fishing mortality rate in region *r* during the time interval [*t*-1,*t*).
$\boldsymbol{m}_{r,t-1}$	Fraction of the available cohort in region *r* killed by all types of fishing mortality (including incidental mortalities) during the time interval [*t*-1,*t*).
Migration Parameters	
$D_{r,t}$	A set of donor regions that contribute fish to region *r* at the end of time period *t*-1 (e.g., used in the PM Model).
$m_{i,j}$	For a given cohort and time, the fraction of the abundance in region *j* moving to region *i*.

Note that the total fishing mortality rate includes both legal and incidental mortalities. Thus, the legal harvest rate is not necessarily equal to the total fishing mortality rate (i.e., $h_{r,t} \neq \boldsymbol{m}_{r,t}$). Also note the following relationships:

$$\boldsymbol{u}_{r,t} = 1 - e^{-M_{r,t}}$$

$$\boldsymbol{m}_{r,t} = 1 - e^{-F_{r,t-1}}$$

$$s_{r,t} = e^{-\left(M_{r,t} + F_{r,t-1}\right)}$$

$$ns_{r,t-1} = n_{r,t-1} s_{r,t}$$

The variable $ns_{r,t-1}$ can be thought of as the number of fish in region *r* at the end of time interval [*t*-1, *t*) just before they are redistributed among the geographic regions for the start of the next time interval [*t*, *t*+1).

C.6.1.3 State Space Model

In matrix notation, the deterministic State Space Model consists of two equations:

$$\mathbf{n}_t = \mathbf{M}_t \mathbf{S}_t \mathbf{n}_{t-1}$$
$$\mathbf{c}_t = \mathbf{H}_t \mathbf{n}_t$$

The abundance vectors $\mathbf{n}_t$ and $\mathbf{n}_{t-1}$ are composed of *R* elements (one abundance for each region). Each migration matrix $\mathbf{M}_t$ is an *R* x *R* square matrix of $m_{i,j}$ elements, and the elements in each column must

sum to one. Each survival matrix $\mathbf{S}_t$ and each harvest matrix $\mathbf{H}_t$ is a diagonal matrix with *R* elements (e.g., $s_{r,t}$, $h_{r,t}$). In expanded form equations (1) and (2) look like this:

$$\begin{bmatrix} n_{1,t} \\ \vdots \\ n_{R,t} \end{bmatrix} = \begin{bmatrix} m_{1,1} & \cdots & m_{1,R} \\ \vdots & & \vdots \\ m_{R,1} & \cdots & m_{R,R} \end{bmatrix} \begin{bmatrix} s_{1,t} & & \\ & \ddots & \\ & & s_{R,t} \end{bmatrix} \begin{bmatrix} n_{1,t-1} \\ \vdots \\ n_{R,t-1} \end{bmatrix}$$

$$\begin{bmatrix} c_{1,t} \\ \vdots \\ c_{R,t} \end{bmatrix} = \begin{bmatrix} h_{1,1} & & \\ & \ddots & \\ & & h_{R,R} \end{bmatrix} \begin{bmatrix} n_{1,t-1} \\ \vdots \\ n_{R,t-1} \end{bmatrix}$$

Note that we can define a new vector $\mathbf{ns}_{r,t-1} = \mathbf{S}_t \mathbf{n}_{t-1}$ that represents the fish that do not suffer natural or fishing mortality during the interval [*t*-1,*t*). Thus, the migration matrix can be thought of as being applied to the surviving cohort at the end of the time interval, and each element of the new abundance vector can be written

$$n_{r,t} = \sum_{j=1}^{R} m_{r,j} ns_{j,t-1} \ .$$

In Ken Newman's application of the State space Model, he sets $M_{r,t} = 0$ for all *r* and *t*. Thus, in his application we have

$$s_{r,t} = e^{-F_{r,t-1}}$$

$$\boldsymbol{u}_{r,t} = 0$$

$$\boldsymbol{m}_{r,t-1} = 1 - e^{-F_{r,t-1}} = 1 - s_{r,t} \ .$$

C.6.1.4 PSC Chinook Model

The PSC Chinook Model defines no specific geographic regions, and therefore has no specific fish migration algorithm. However, by separating the 25 model fisheries into preterminal and terminal categories, the model assumes a de facto concept of fish migration from the preterminal area to the terminal area. The preterminal fisheries are generally offshore ocean fisheries (e.g., ocean troll fisheries), while terminal fisheries are generally nearshore and river fisheries (e.g., ocean net and sport fisheries).

Once the preterminal fishery harvests and incidental mortalities have been taken, the model separates each cohort into an ocean run (that stays in the ocean and is available for harvest the following year) and a terminal run (that is available for harvest by terminal fisheries). Thus, what are termed maturation rates in the PSC Chinook Model can be thought of as migration rates from the ocean (preterminal area) to the nearshore and river areas (collectively called the terminal area).

Some ocean net fisheries are considered to be terminal fisheries for older age classes, even though these fisheries are not physically located near the natal stream. The idea is that at some point during the year the older (mature) members of each cohort decide to start migrating down the coast. It is assumed that the older fish captured by the nearshore net fisheries are part of the mature migrating portion of the stock.

Using the mathematical notation of the PSC Chinook Model, the maturation rates are applied to each cohort as follows:

$$TermRun = (OcnRun - PreTermMort) \cdot MatRt$$

where

TermRun = the abundance of fish in the terminal area at the start of the terminal time period;

OcnRun = the abundance of fish in the preterminal area during the preterminal time period after natural mortalities have been removed;

PreTermMort = total fishing mortalities (i.e., legal catches plus incidental mortalities) in the preterminal area during the preterminal time period.

The term *"OcnRun* - *PreTermMort"* is the abundance of fish not suffering natural mortality and surviving the fishing process during the preterminal time period. In the notation of the State Space Model, this term is equivalent to the first element of the $\mathbf{ns}_{t-1}$ vector. The second element of the $\mathbf{ns}_{t-1}$ vector is zero because it is assumed that no fish are located in the terminal area during the preterminal time period.

The term "*TermRun"* is equivalent to the second element of the $\mathbf{n}_t$ vector (i.e., abundance in region 2 at the start of time period 2). The first element of the $\mathbf{n}_t$ vector is the number of fish from the cohort that remain in the preterminal (or ocean) area.

In matrix notation the annual migration pattern for each cohort is expressed as follows (where 1 = preterminal area and time step; 2 = terminal area and time step):

$$\begin{bmatrix} n_{1,2} \\ n_{2,2} \end{bmatrix} = \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} \cdot \begin{bmatrix} ns_{1,1} \\ ns_{2,1} \end{bmatrix} = \begin{bmatrix} m_{1,1}ns_{1,1} + m_{1,2}ns_{2,1} \\ m_{2,1}ns_{1,1} + m_{2,2}ns_{2,1} \end{bmatrix}$$

where

$n_{1,2}$ = preterminal area (i.e., ocean) abundance at the start of the terminal time step

$n_{2,2}$ = terminal area abundance at the start of the terminal time step

$m_{1,1}$ = 1 - *MatRt*

$m_{1,2}$ = 0 (i.e., no fish move from the terminal area to the preterminal area)

$m_{2,1}$ = *MatRt*

$m_{2,2}$ = 0 (i.e., there are no fish in the terminal area to keep in the terminal area)

$ns_{1,1}$ = preterminal area (i.e., ocean) abundance at the end of the preterminal time step

$ns_{2,1}$ = 0 (i.e., no fish in the terminal area at the end of the preterminal time step)

These equations are now in the same form as Equation (3) for the State Space Model. In terms of the maturation rate we have

$$n_{1,2} = (1 - MatRt) \cdot ns_{1,1}$$

$$n_{2,2} = MatRt \cdot ns_{1,1} \;.$$

C.6.1.5 PSC Selective Fisheries Model

This model can have any number of marine geographic areas, but in its initial application contains just five:

- West Coast Vancouver Island (OCNN);
- Washington/Oregon Ocean (OCNS);
- Strait of Georgia (GEOS);
- Strait of Juan de Fuca and San Juan Islands (SJDF);
- South Puget Sound (SSND).

Some stocks also have terminal areas (e.g., CRTM = Columbia River terminal area) and all stocks have a generic escapement, or spawning, area (ESCP).

This model was designed primarily as a coho model in which all members of a cohort (or brood) mature and return to the natal stream in the same year. The model is generally used to examine the potential effects of selective fishery management on a limited number of stocks during a single year. Thus, this model is not designed to simulate the effects of coastwide management regulations on all stock and fisheries.

This model can be run in either deterministic or stochastic mode. The model flow is as follows:

1. Compute initial abundance of each stock;
2. Distribute initial abundance to the five fishing areas;
3. Time loop with:
    a. Natural mortality;
    b. Fishing mortality;
    c. Redistribute the fish (including escapement).

Several simplifying assumptions were made to estimate the proportion of each stock in each area migrating in each time step:

1. No migration occurred between geographic regions during statistical weeks 1 through 32;
2. All migration was directed toward the river of origin;
3. Hatchery and wild fish had the same migration timing and pathway;
4. 33% of InStk1 in the OCNN region migrated around the north end of Vancouver Island to the GEOS region; and
5. Catch per unit effort in a fishery provided an unbiased measure of stock abundance.

Mathematically, the migration component of this model is virtually identical in structure to the State Space Model. During each time step, natural and fishing mortalities (including incidental mortalities) are removed from the cohort first. The remaining fish in a given area are then redistributed among the areas by assigning "dispersion" rates to each area. In the notation of the PSC Selective Fishery Model, the new abundances are computed by the following equation:

$$N_{s,a',t} = \sum_{a=1}^{A} I^{*}_{s,a,a',t}$$

where

$N_{s,a,t}$ = the abundance of stock s in area a at time t;

$I_{s,a,b,t}$ = the number of fish from stock s that move from area a to area b at time t.

In stochastic mode, the elements of the migration matrix are selected randomly from a multinomial distribution with parameters:

$\xi_{s,a,b,t}$ = probability that a fish from stock *s* moves from area *a* to area *b* at time *t*

These probabilities are estimated from catch and effort data using a "solver" routine in Microsoft Excel. Table 9 lists the dispersion parameters. In deterministic mode, the probabilities are replaced by the fractions moving from one region to another. Thus, in the PSC Selective Fishery Model the $\xi_{s,a,b,t}$ are equivalent to the $m_{i,j}$ elements of the migration matrix of the state space model.

Table 9 Dispersion parameters by week for the PSC Selective Fishery Model.

Stock	Donor	Receiver	32	33	34	35	36	37	38	39	40	41	42
WCVI	OCNN	ESCP	0.00	0.01	0.03	0.06	0.07	0.17	0.14	0.38	0.26	0.40	0.47
WCVI	SJDF	OCNN	0.17	0.19	0.16	0.17	0.13	0.13	0.10	0.12	0.16	0.24	0.35
LFGS	OCNN	SJDF	0.00	0.00	0.01	0.09	0.15	0.23	0.15	0.39	0.59	0.86	0.89
LFGS	OCNS	SJDF	0.00	0.12	0.12	0.28	0.30	0.56	0.40	0.26	0.00	0.00	0.00
LFGS	SJDF	GEOS	0.00	0.00	0.00	0.00	0.00	0.00	0.15	0.22	0.31	0.24	0.31
LFGS	GEOS	ESCP	0.00	0.00	0.00	0.05	0.05	0.11	0.11	0.16	0.16	0.29	0.29
NPSD	OCNN	SJDF	0.01	0.00	0.02	0.13	0.22	0.33	0.22	0.47	0.34	0.36	0.03
NPSD	OCNS	SJDF	0.00	0.07	0.07	0.23	0.30	0.62	0.52	0.43	0.11	0.38	0.67
NPSD	GEOS	SJDF	0.00	0.00	0.00	0.00	0.00	0.01	0.05	0.05	0.06	0.07	0.18
NPSD	SJDF	NSND	0.00	0.01	0.01	0.02	0.01	0.05	0.09	0.20	0.28	0.35	0.34
NPSD	NSND	ESCP	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.67	1.00	1.00	1.00
SPSD	OCNN	SJDF	0.00	0.00	0.00	0.12	0.20	0.31	0.19	0.44	0.67	0.98	0.98
SPSD	OCNS	SJDF	0.00	0.07	0.07	0.23	0.30	0.63	0.71	0.79	0.79	0.89	1.00
SPSD	GEOS	SJDF	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.07	0.15
SPSD	SJDF	SSND	0.00	0.00	0.00	0.02	0.03	0.03	0.10	0.12	0.21	0.16	0.29
SPSD	SSND	ESCP	0.00	0.00	0.00	0.00	0.00	0.33	0.33	0.67	0.54	0.87	0.87
NWAC	OCNN	OCNS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.24	0.40	0.40	0.50
NWAC	OCNS	WCTM	0.00	0.00	0.00	0.09	0.22	0.43	0.68	0.88	1.00	1.00	1.00
NWAC	SJDF	OCNS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.14	0.27
NWAC	WCTM	ESCP	0.01	0.00	0.34	0.33	0.33	0.00	0.00	0.00	0.00	0.14	0.14
CRIV	OCNS	CRTM	0.00	0.06	0.10	0.21	0.42	0.38	0.60	0.65	0.99	0.99	1.00
CRIV	SJDF	OCNS	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.12	0.22	0.11	0.31
CRIV	CRTM	ESCP	0.10	0.10	0.19	0.17	0.17	0.07	0.00	0.00	0.29	0.40	0.40

Table 9 Concluded.

Stock	Donor	Receiver	43	44	45	46	47	48	49	50	51	52
WCVI	OCNN	ESCP	0.81	1.00	1.00	1.00	1.00					
WCVI	SJDF	OCNN	0.60	0.82	0.99	0.99	1.00					
LFGS	OCNN	SJDF	0.94	1.00								
LFGS	OCNS	SJDF	0.00	0.00	0.00	0.00	0.00	0.05	0.12	0.46	0.74	1.00
LFGS	SJDF	GEOS	0.39	0.31	0.28	0.13	0.14	0.03	0.11	0.11	0.44	1.00
LFGS	GEOS	ESCP	0.45	0.45	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
NPSD	OCNN	SJDF	0.03	0.35	0.67	1.00						
NPSD	OCNS	SJDF	1.00									
NPSD	GEOS	SJDF	0.27	0.23	0.12	0.00	0.00	0.00	0.00	0.00	0.00	1.00
NPSD	SJDF	NSND	0.31	0.25	0.21	0.17	0.11	0.09	0.06	0.09	0.12	1.00
NPSD	NSND	ESCP	1.00	1.00	1.00	0.73	0.73	0.73	0.76	0.76	0.49	1.00
SPSD	OCNN	SJDF	0.98	1.00								
SPSD	OCNS	SJDF										
SPSD	GEOS	SJDF	0.29	0.25	0.16	0.00	0.00	0.00	0.00	0.00	0.33	1.00
SPSD	SJDF	SSND	0.32	0.41	0.32	0.20	0.07	0.02	0.15	0.15	0.46	1.00
SPSD	SSND	ESCP	1.00	0.76	0.50	0.30	0.20	0.13	0.00	0.22	0.56	1.00
NWAC	OCNN	OCNS	0.67	1.00								
NWAC	OCNS	WCTM	0.67	0.67	0.67	0.67	0.33	0.33	0.67	1.00	1.00	1.00
NWAC	SJDF	OCNS	0.21	0.24	0.23	0.23	0.12	0.03	0.07	0.17	0.47	1.00
NWAC	WCTM	ESCP	0.17	0.13	0.15	0.18	0.08	0.19	0.27	0.57	0.78	1.00
CRIV	OCNS	CRTM	1.00	1.00	1.00	1.00						
CRIV	SJDF	OCNS	0.44	0.76	0.90	1.00						
CRIV	CRTM	ESCP	0.11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	1.00

The following example illustrates the relationship between the $\varepsilon_{s,a,b,t}$ and the $m_{i,j}$. Table 10 lists the dispersion rate (and "Non-Dispersion Rate") parameters for the South Puget Sound stock during week 40.

Table 10 Dispersion and non-dispersion (= 1 - dispersion) rate parameters for the South Puget Sound stock during week 40 used in the PSC Selective Fishery Model.

Donor Region	Receiving Region	Dispersion Rate	Non-Dispersion Rate
Ocean North	Strait Juan de Fuca	0.67	0.33
Ocean South	Strait Juan de Fuca	0.79	0.21
Georgia Strait	Strait Juan de Fuca	0.03	0.97
Strait Juan de Fuca	South Puget Sound	0.21	0.79
South Puget Sound	Escapement	0.54	0.46

It is assumed that fish that do not disperse from a donor region remain in the donor region. There are six areas counting the "Escapement" area (1 = OCNN; 2 = OCNS; 3 = SJDF; 4 = GEOS; 5 = SSND; and 6 = ESCP). The corresponding State Space Model migration matrix representing this movement pattern is:

$$M_{40} = \begin{bmatrix} 0.33 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.21 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.97 & 0 & 0 & 0 \\ 0.67 & 0.79 & 0.03 & 0.79 & 0 & 0 \\ 0 & 0 & 0 & 0.21 & 0.46 & 0 \\ 0 & 0 & 0 & 0 & 0.54 & 1.00 \end{bmatrix}$$

Note that values along the diagonal represent the non-dispersion rates (i.e., the fraction of the cohort that remains in the region).

C.6.1.6 Proportional Migration (PM) Model

The PM model does not have specific geographic objects. However, fishery definitions act as de facto geographic objects. For example, the sport fisheries are generally defined by statistical reporting area, such as "Area 7", "Area 8", "Buoy 10", etc. There are 45 fisheries/geographic areas in this model.

There are three gear types used to define fisheries and each gear type partitions the geographic range differently. For example, within the troll gear group there is a Northwest Vancouver Island and a Southwest Vancouver Island and within the sport gear group there is only one fishery for all of the West Coast of Vancouver Island. This methodology assumes that fish available for the WCVI sport fishery are not the same fish that are also available for the two WCVI troll fisheries. In other words, the model tacitly assumes that there are three distinct geographic regions off WCVI. In terms of the notation used in this report, the subscript *r* indexes both geographic regions and their associated fisheries.

There is no explicit movement of fish between fishing regions. However, the fisheries are assumed to have a geographic and temporal ordering such that the fish available in a given fishery and time are received only from designated donor fishing regions (designated by the set $D_{r,t}$). Note that several receiving regions may share some of the donor regions. For example, region 2 may receive fish from regions 1 and 2 and region 3 may receive fish from regions 2 and 3.

The model expresses changes in fishery mortalities relative to a base year. For a given base year and stock, the model begins with input data for three parameters (obtained from run reconstructions): $c_{r,t}$, $h_{r,t}$ and $E_{r,t}$. The harvest rates ($h_{r,t}$) are assumed to be fishery and time specific, but not stock specific. That is, the same $h_{r,t}$ term is applied to all stocks harvested in region (fishery) *r* at time *t*.

The region and time specific abundances at the start of time interval [*t*-1,*t*) are computed by

$$n_{r,t-1} = \frac{c_{r,t}}{h_{r,t}} \cdot EV$$

where *EV* is a stock and year specific "Environmental Variability" scalar. For the base year *EV* = 1. For years other than the base year, these initial abundances are adjusted by the *EV* scalars to reflect the brood year strength relative to the base year. These scalars have the same effect as *EV* scalars in the PSC chinook model.

Note that the regional abundances are determined solely from the input data for that region and time step. Thus, it is possible for the total abundance for a stock to be greater at time *t* than at time *t*-1. That is,

$$\sum_r n_{r,t} > \overset{?}{=} < \sum_r n_{r,t-1} .$$

The survivors after the fishing process during the period [*t*-1,*t*) are computed by

$$ns_{r,t-1} = n_{r,t-1} - c_{r,t}$$

The model simulates the impacts of changes in region specific harvest rates as follows (primes in the notation indicate variables under the new harvest management scenario). Let $E'_{r,t}$ be the new effort level in region *r* at time *t*. Let the associated new harvest rate be some function of the base period effort, base period harvest rate, and the new effort level. That is,

$$h'_{r,t} = f\left(h_{r,t}, E_{r,t}, E'_{r,t}\right)$$

This function may be a simple linear scaling, such as

$$h'_{r,t} = h_{r,t} \cdot \frac{E'_{r,t}}{E_{r,t}}$$

In the first time step the new harvest rate is applied to the original abundance ($n_{r,0}$) to get the new catch:

$$c'_{r,1} = h'_{r,1} n_{r,0}$$

New survivors in the first time step are computed by

$$ns'_{r,1} = n_{r,0} - c'_{r,1}$$

Thus, during the first time step we have

$$c_{r,1} = h_{r,1} n_{r,0}$$

$$c'_{r,1} = h'_{r,1} n_{r,0}$$

and we can write

$$\frac{c_{r,1}}{h_{r,1}} = \frac{c'_{r,1}}{h'_{r,1}} .$$

Solving for the new catch we get

$$c'_{r,1} = c_{r,1} \frac{h'_{r,1}}{h_{r,t}}$$

Thus, the new regional catches in the first time step are just the old regional catches scaled up or down by the ratio of the new and old harvest rates. Note that the scaling is based on the relative harvest rates, not the effort levels. If a non-linear relationship is used to relate effort and harvest rate, doubling the effort may not necessarily double the harvest rate and the catch.

In subsequent time steps, the new harvest rate is not applied to the original local abundance. Instead, the original local abundance is scaled up or down to reflect the total changes in fishing mortality in the donor regions during the previous time step. The scaling factor is the ratio of the new survivors to original survivors (from the donor areas) during the previous time step, and the new abundance is computed as follows:

$$n'_{r,t} = n_{r,t} \cdot \frac{\sum_{r \in D_{r,t}} ns'_{r,t-1}}{\sum_{r \in D_{r,t}} ns_{r,t-1}}$$

where the new survivors at time *t*-1 are computed by

$$ns'_{r,t-1} = n'_{r,t-1} - c'_{r,t} \, .$$

If we separate the original (base) variables from the adjusted variables we can write

$$n'_{r,t} = \frac{n_{r,t}}{\sum_{r \in D_r} ns_{r,t-1}} \cdot \sum_{r \in D_r} ns'_{r,t-1}$$

Note that this equation is quite similar to the equation for the elements of the new abundance vector in the State Space Model formulation, namely:

$$n_{r,t} = \sum_{j=1}^{R} m_{r,j} ns_{j,t-1}$$

Recall that $m_{r,j}$ is the fraction of the fish located in donor region *j* moving into region *r*. If one assumes a constant migration rate during the interval [*t*-1,*t*) from all donor regions *j* into region *r* (call the constant $K_{r,t} = m_{r,j}$ for all *j*), the migration rate terms can be moved outside the summation sign:

$$n_{r,t} = K_{r,t} \sum_{j=1}^{R} ns_{j,t-1}$$

For the PM model, we can set

$$K_{r,t} = \frac{n_{r,t}}{\sum_{r \in D_r} ns_{r,t-1}} \, .$$

In the PM model the term $K_{r,t}$ must be thought of as a migration index, rather than a migration rate. It is the ratio of the fish located in region *r* at the start of time period *t* to the total potential donor fish for region

r at the end of time period t-1. Since the abundances for different time periods are computed independently, this ratio can be greater than one and does not represent a movement of fish from one area to another.

Regardless of the biological interpretation of the $K_{r,t}$ terms, the important point is that the mathematical computations of the PM Model can be conducted using the matrix formulation of the State Space Model by substituting the $K_{r,t}$ terms in the appropriate position of the migration matrix. For time t, each $K_{r,t}$ term is placed in each column of the rth row that corresponds to a donor region for region r.

The State Space Model formulation of the PM model is best explained with an example. Consider a case of three fisheries (regions) and two time steps. Assume that for the second time step the donor regions are as follows:

Receiving Region	Donor Regions
1	1
2	1 and 2
3	2 and 3

Then from the input data we have:

$$K_{1,2} = \frac{n_{1,2}}{ns_{1,1}}$$

$$K_{2,2} = \frac{n_{2,2}}{ns_{1,1} + ns_{2,1}}$$

$$K_{3,2} = \frac{n_{3,2}}{ns_{2,1} + ns_{3,1}}$$

In terms of the matrix computations we have

$$\begin{bmatrix} n'_{1,2} \\ n'_{2,2} \\ n'_{3,2} \end{bmatrix} = \begin{bmatrix} K_{1,2} & 0 & 0 \\ K_{2,2} & K_{2,2} & 0 \\ 0 & K_{3,2} & K_{3,2} \end{bmatrix} \begin{bmatrix} ns'_{1,1} \\ ns'_{2,1} \\ ns'_{3,1} \end{bmatrix}$$

where the $\mathbf{ns}'_1$ abundance vector represents the fish surviving the new harvest regime during the first time step. In practice $\mathbf{ns}'_1$ vector would be computed by multiplying a new survival matrix (with the diagonal elements representing the new survivals determined from the original and new effort levels) times the original abundance vector.

The specific element formulas for the $\mathbf{n}'_2$ vector are:

$$n'_{1,2} = n_{1,2} \cdot \frac{ns'_{1,1}}{ns_{1,2}}$$

$$n'_{2,2} = n_{2,2} \cdot \frac{ns'_{1,1} + ns'_{2,1}}{ns_{1,1} + ns_{2,1}}$$

$$n'_{3,2} = n_{3,2} \cdot \frac{ns'_{2,1} + ns'_{3,1}}{ns_{2,1} + ns_{3,1}}$$

To compare these values with values determined from a migration matrix we rearrange terms to get:

$$n'_{1,2} = \frac{n_{1,2}}{ns_{1,2}} \cdot ns'_{1,1} = K_{1,1} \cdot ns'_{1,1}$$

$$n'_{2,2} = \frac{n_{2,2}}{ns_{1,1} + ns_{2,1}} \cdot ns'_{1,1} + \frac{n_{2,2}}{ns_{1,1} + ns_{2,1}} ns'_{2,1} = K_{2,2} \cdot \left(ns'_{1,1} + ns'_{2,1}\right)$$

$$n'_{3,2} = \frac{n_{3,2}}{ns_{2,1} + ns_{3,1}} ns'_{2,1} + \frac{n_{3,2}}{ns_{2,1} + ns_{3,1}} ns'_{3,1} = K_{3,2} \cdot \left(ns'_{2,1} + ns'_{3,1}\right)$$

The corresponding values assuming a given migration matrix are:

$$n'_{1,2} = m_{1,1} \cdot ns'_{1,1}$$

$$n'_{2,2} = m_{2,1} \cdot ns'_{1,1} + m_{2,2} \cdot ns'_{2,1}$$

$$n'_{3,2} = m_{3,2} \cdot ns'_{2,1} + m_{3,3} \cdot ns'_{3,1}$$

If the terms in the transition matrix represent the true movement of fish between areas, then it is clear that the new regional abundances (due to a different fishing regime) predicted by the PM model are correct only under very limited circumstances. To see the relationship between the true migration rates and the *K* terms we set the new abundance values equal and solving for the *K* terms to get:

$$K_{1,1} = m_{1,1}$$

$$K_{2,2} = \frac{ns'_{1,1}}{\left(ns'_{1,1} + ns'_{2,1}\right)} \cdot m_{2,1} + \frac{ns'_{2,1}}{\left(ns'_{1,1} + ns'_{2,1}\right)} \cdot m_{2,2}$$

$$K_{3,2} = \frac{ns'_{2,1}}{\left(ns'_{2,1} + ns'_{3,1}\right)} \cdot m_{3,2} + \frac{ns'_{3,1}}{\left(ns'_{2,1} + ns'_{3,1}\right)} \cdot m_{3,3}$$

The PM model will give correct predictions only when the above relationships are true.

In conclusion, it is possible to cast the PM model in the general matrix framework as follows:

- Use the regional abundances at the start of the first time step for the initial abundance vector;
- Use the input catch and harvest rate data to compute the $K_{r,t}$ terms;
- Use the input and adjusted effort data to create new survival matrices;
- Place the $K_{r,t}$ terms in the appropriate location in the migration matrices;
- Perform the matrix computations in chronological order.

It is clear from the above formulae that the PM model makes the tacit assumption that for a given cohort and time step, fish migrate from all donor regions *at the same rate*. This is very different from the PSC Selective Fishery Model that estimates separate migration rates for each donor area. Recall from the PSC example for week 40 for the SPSD stock that fish migrate into the SJDF area from four other areas at very

different rates: OCCN = 0.67; OCNS = 0.79; GEOS = 0.03; and SJDF = 0.79. The PM model would require that fish would enter the SJDF area at the same rate from each of the four areas.

To clearly see the relationship between the estimated abundances used by the PM model and those determined

Assume the matrix M represents the true movement of a cohort between areas and let h

We also note that if the set of donor regions for each area and time step ($D_{r,t}$) is the entire set of possible regions, the PM model is virtually identical to a single pool model in which the original regional harvest rates are defined with respect to the total abundance of the cohort at any given time (instead of local abundance within a region). That is, under a single pool model the pooled harvest rate ($ph_{r,t}$) is defined as:

$$ph_{r,t} = \frac{c_{r,t}}{\sum_r n_{r,t-1}}.$$

The catch equation is:

$$c_{r,t} = ph_{r,t} \sum_r n_{r,t-1}.$$

The base regional catches are the same under both formulations of the harvest rate (PM and Single Pool), so we can write:

$$ph_{r,t} \sum_r n_{r,t-1} = h_{r,t} n_{r,t-1}$$

and

$$ph_{r,t} = h_{r,t} \frac{n_{r,t-1}}{\sum_r n_{r,t-1}}$$

Thus, the base harvest rate in a single pool model is equal to the base local harvest rate in the PM model scaled by the fraction of the total cohort abundance located in the region.

Recall that during the first time step of the PM model, the local abundances are not adjusted before applying the adjusted harvest rates and the adjusted catches are just scaled by the ratio of the new to the old harvest rate:

$$c'_{r,1} = c_{r,1} \frac{h'_{r,1}}{h_{r,t}}$$

For the single pool model, the same is true. Here we have

$$c_{r,1} = ph_{r,1} \sum_r n_{r,0}$$

$$c'_{r,1} = ph'_{r,1} \sum_r n_{r,0}$$

Thus, we can write

$$\frac{c_{r,1}}{ph_{r,1}} = \frac{c'_{r,1}}{ph'_{r,1}}$$

and

$$c'_{r,1} = c_{r,1} \frac{ph'_{r,1}}{ph_{r,t}}$$

Thus, at the first time step the regional catches are scaled up and down to reflect the changes in the harvest rates. If the set of donor regions for each area and time step is the entire set of possible regions, then the same will be true for all time steps.

C.6.1.7 Fishery Resource Allocation Model (FRAM)

This model is similar to the PSC Chinook Model in that it has no specific geographic areas, but does partition the fisheries into preterminal and terminal categories, and also includes separate extreme terminal areas. This model uses monthly time steps. At each time step the maturation algorithm is called, but it is unclear whether or not a maturation calculation actually occurs at each time step. Need further clarification on this.

The Terminal Area Management Modules (TAMMs) used in FRAM do not have any migration components. The harvest rates in the extreme terminal areas are all defined with respect to the abundance of fish entering the extreme terminal area.

C.6.2 Beta Advection-Diffusion Model

C.6.2.1 Background

The initial migration sub-model used in Ken Newman's State Space Model (SSM) was loosely called a Beta Advection-Diffusion model. The SSM estimates a single parameter (*Move_Alpha*) that completely describes the movement of fish over all time periods and locations. At the August 27, 1998 model committee meeting many members asked for an intuitive description of how this single parameter characterized fish migration. The purpose of this note is to define the Beta Advection-Diffusion model and describe its behavior. Another description is given in Ken's latest draft of his paper.

C.6.2.2 The Model

Let l_t be the current location of a fish (at time t). Then the next location of that fish (at time $t + 1$) is described by a Beta($\boldsymbol{a}$,$\boldsymbol{b}$) probability distribution where $\boldsymbol{b}$ is assumed constant (= 3.0) and $\boldsymbol{a}$ is a function of *Move_Alpha*, current location (l_t), and current time (t) as follows:

$$a = (Move_Alpha) \cdot (Dist_Scalar) \cdot (Time_Scalar)$$

where the distance scalar (*Dist_Scalar*) represents how close the fish is to the natal stream and is given by

$$Dist_Scalar = \frac{l_t}{D}$$

(D = maximum possible distance from the natal stream) and the time scalar (*Time_Scalar*) represents how close the current time is to the last time period (T) and is given by

$$Time_Scalar = \frac{T+1-t}{T+1}.$$

The "1" in the numerator prevents a zero value for *Time_Scalar* and the "1" in the denominator roughly offsets the "1" in the numerator. The net effect is that as a fish moves closer to the natal stream (l_t gets smaller) and as time increases, both scalars get smaller and the $\boldsymbol{a}$ parameter of the Beta distribution gets smaller.

C.6.2.3 Model Properties

The expected value of a Beta distributed random variable is $\boldsymbol{a}/(\boldsymbol{a}+\boldsymbol{b})$. Thus, as time increases and the $\boldsymbol{a}$ parameter gets smaller, the expected new location moves toward the natal stream.

The expected step size for a fish located at l_t and time t is

$$E(Step) = l_t - \frac{\boldsymbol{a}}{\boldsymbol{a}+\boldsymbol{b}}.$$

Since $\boldsymbol{b}$ is fixed and $\boldsymbol{a}$ is a function of *Move_Alpha*, l_t, and t, expected step size can be computed for all possible combinations of location and time. Or more generally, one can compute expected step size as a function of *Dist_Scalar* and *Time_Scalar*.

To get a better "feel" for this migration model, I created an Excel Spreadsheet to compute a table of expected step sizes for combinations of *Dist_Scalar* (0 to 1 in 0.1 increments) and *Time_Scalar* (also 0 to 1 in 0.1 increments) given *Move_Alpha* and $\boldsymbol{b}$ (Table 11). I plotted the resulting expected step size surface as a 3D plot (Figure 1) and also plotted expected step size as a function of current location for five relative time values (0.1, 0.3, 0.5, 0.7, and 0.9;Figure 2). The table values and plots update automatically whenever *Move_Alpha* and $\boldsymbol{b}$ are changed.

The general migration pattern is the following: at any given time, there is an "Attracting Location" at which the expected step size is zero. Fish located to the right of this location (i.e., Current Location > Attracting Location) have a negative expected step size (i.e., they move toward the Attracting Location); fish located to the left of this location (i.e., Current Location < Attracting Location) have a positive expected step size (i.e., they also move toward the Attracting Location). As the season progresses (i.e., time gets bigger), the Attracting Location moves in the direction of the natal stream (i.e., the origin).

With $\boldsymbol{b}$ fixed at 3.0, increasing values of *Move_Alpha* move the initial Attracting Location further from the origin. I created a table (Table 12) and graph (Figure 3) to illustrate the relationship between Attracting Location and time for values of *Move_Alpha* ranging from 3.0 to 15.0. When *Move_Alpha* is about 5.0, the Attracting Location is near the origin for all times. When *Move_Alpha* = 9.0, the Attracting Location starts out at about 0.67 and when current time reaches about 0.67 the Attracting Location is the origin. When *Move_Alpha* = 15.0, the Attracting Location starts out at about 0.80 and when current time reaches 0.80 the Attracting Location is the origin.

C.6.2.4 Discussion

What follows is a summary of email correspondence between Jim Norris and Ken Newman.

Norris: Although the Beta Advection-Diffusion model has some appealing properties (e.g., Attracting Location moves toward the origin as time increases), I find the property of an Attracting Location early in the season somewhat disturbing because it implies a directed movement toward a specific location other than the natal stream. And for coastal stocks, such as the Humptulips, it implies that fish located north and

south of the natal stream during the first part of the modeling period have directed movement toward different locations (i.e., there are two Attracting Locations, one north and one south of the natal stream).

Newman: Ken offered another interpretation. Early in the season, fish near the natal area are "free" to keep moving away from the natal area ...they're closing enough that they've got time to "dawdle," while fish further away need to get moving home.

Norris: I'm having difficulty linking this type of movement model with my intuition about how salmon behave in the ocean. I think the initial fish distribution is determined by a combination of genetic factors and physical and biological oceanographic conditions. The genetic factors seem to put limits on the range of latitudes the fish are willing to inhabit, while oceanographic conditions determine the degree of patchiness, or aggregations, within those limits (e.g., due to prey, predators, temperature, salinity, olfactory cues, etc). Thus, I think it is possible for the initial distribution of a stock to have one or more concentrations. In our modeling effort I think we need a flexible distribution to account for skewed initial distributions.

Newman: Ken noted that another factor affecting initial fish distribution is size and time of release for hatchery fish. This also affects maturation rates for chinook. With the Humptulips stocks apparently "turning" south and north--a bimodal initial distribution could make some sense.

Norris: Given the initial distribution of the fish, I think the migration model should be founded, as much as possible, on what is known about individual fish behavior. Unfortunately, not much is known! And since we all have our own ideas about what that behavior is, I think we'll have to try several types of models to see which ones fit real data the best, and also make the most sense biologically. As you mentioned in one of your emails, the fact that the Beta Advection-Diffusion model predicts some unrealistically high migration rates late in the season may not be a problem because those unrealistic rates are only predicted for fish residing outside the realistic domain of the fish at that time. The same is true for the "Increasing" migration model I used to generate synthetic data--by the time the last time steps are reached, the daily rate is pretty high, but all the fish are already in the river.

Newman: Your key statement is "not much is known" -- which has kept me pointed towards doing largely data driven selection of models. If someone comes up with an alternative "module" for initial distribution, survival/harvest, and/or migration and it fits better for a relatively large set of stocks and cohorts than does a current configuration, then that's as good an argument as any to pick the alternative.

Table 11 Expected Step Size for values of relative location and time (*Move_Alpha* = 8.735; ? = 3.0).

Rel	Relative Elapsed Time										
Loc	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.1	0.13	0.11	0.09	0.07	0.05	0.03	0.00	-0.02	-0.04	-0.07	-0.10
0.2	0.17	0.14	0.12	0.09	0.06	0.03	-0.01	-0.05	-0.10	-0.14	-0.20
0.3	0.17	0.14	0.11	0.08	0.04	0.00	-0.04	-0.09	-0.15	-0.22	-0.30
0.4	0.14	0.11	0.08	0.05	0.01	-0.03	-0.08	-0.14	-0.21	-0.30	-0.40
0.5	0.09	0.07	0.04	0.00	-0.03	-0.08	-0.13	-0.20	-0.27	-0.37	-0.50
0.6	0.04	0.01	-0.02	-0.05	-0.09	-0.13	-0.19	-0.26	-0.34	-0.45	-0.60
0.7	-0.03	-0.05	-0.08	-0.11	-0.15	-0.20	-0.25	-0.32	-0.41	-0.53	-0.70
0.8	-0.10	-0.12	-0.15	-0.18	-0.22	-0.26	-0.32	-0.39	-0.48	-0.61	-0.80
0.9	-0.18	-0.20	-0.22	-0.25	-0.29	-0.33	-0.39	-0.46	-0.56	-0.69	-0.90
1.0	-0.26	-0.28	-0.30	-0.33	-0.36	-0.41	-0.46	-0.53	-0.63	-0.77	-1.00

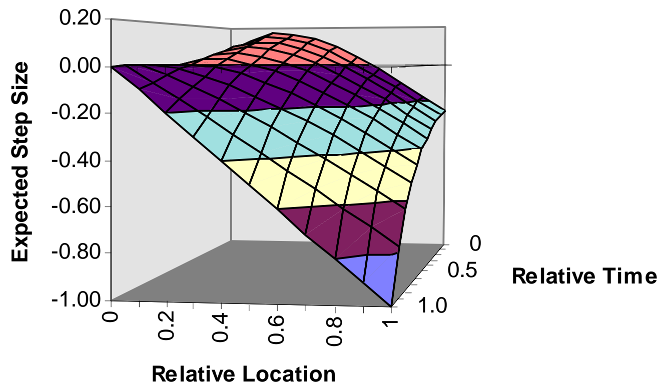


Figure 1 Expected step size as a function of relative location (i.e., *Dist_Scalar*) and relative time (*Time_Scalar*).

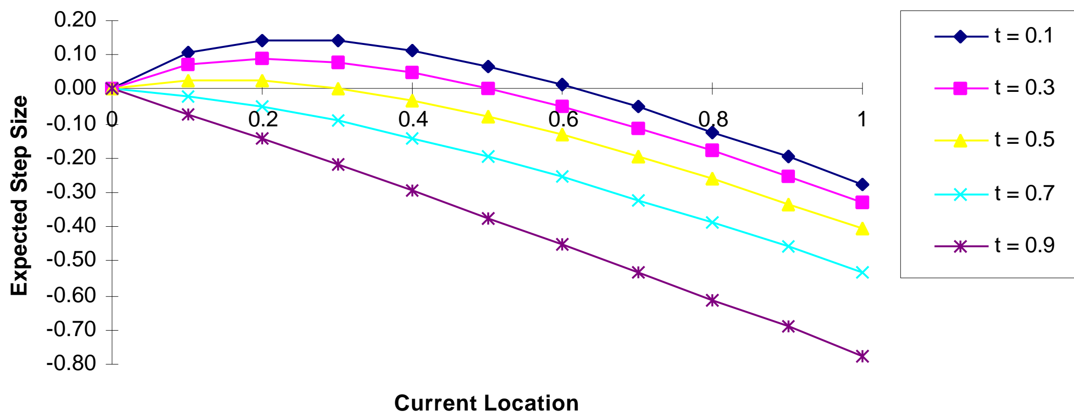


Figure 2 Expected step size as a function of current location for five values of relative t.

Table 12 "Attraction Location" by Relative Time and *Move_Alpha*.

	Move_Alpha						
Rel Time	3.00	5.00	7.00	9.00	11.00	13.00	15.00
0	0.00	0.40	0.57	0.67	0.73	0.77	0.80
0.1	-0.11	0.33	0.52	0.63	0.70	0.74	0.78
0.2	-0.25	0.25	0.46	0.58	0.66	0.71	0.75
0.3	-0.43	0.14	0.39	0.52	0.61	0.67	0.71
0.4	-0.67	0.00	0.29	0.44	0.55	0.62	0.67
0.5	-1.00	-0.20	0.14	0.33	0.45	0.54	0.60
0.6	-1.50	-0.50	-0.07	0.17	0.32	0.42	0.50
0.7	-2.33	-1.00	-0.43	-0.11	0.09	0.23	0.33
0.8	-4.00	-2.00	-1.14	-0.67	-0.36	-0.15	0.00
0.9	-9.00	-5.00	-3.29	-2.33	-1.73	-1.31	-1.00

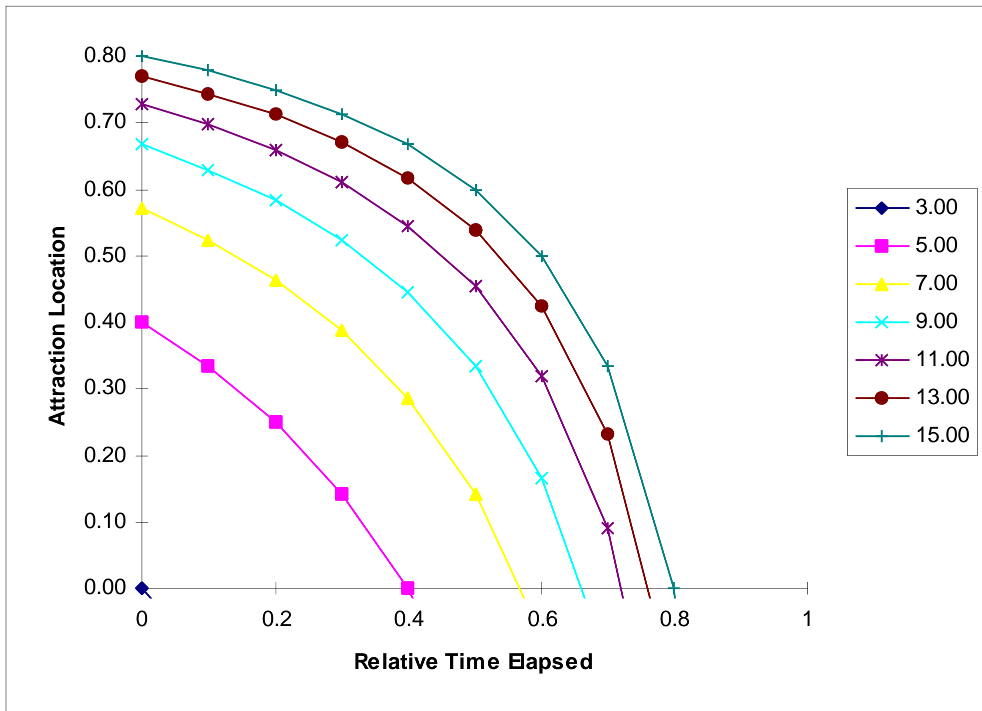


Figure 3 "Attraction Location" as a function of relative time for several values of *Move_Alpha*.