A simulation framework for developing optimal sampling strategies for the Maine sea urchin stock

A report submitted to the Northeast Consortium and Maine Department of Marine Resources

Computer simulation work is funded by the Northeast Consortium Program Development fund (Gant #: 02-628)

Field sampling and data collection are partially funded by a DMR Marine Science Graduate fellowship (Grant #: G601240)

by

Robert Grabowski[1,a], Yong Chen[a,c], Robert Russell[b], and Margaret Hunter[b]

[a] School of Marine Sciences, University of Maine, Orono, ME 04469

[b] The Maine Dept. of Marine Resources, West Boothbay Harbor, Maine

[c] Contact: ychen@maine.edu; Tel: (207) 581-4303; Fax: (207) 581-4388

May 14, 2003

[1] This report consists of part of MS thesis of Robert Grabowski

Executive summary

Fishery-independent surveys are scientific studies that provide essential information for stock assessments and for developing appropriate management plans. Pilot studies are conducted prior to the start of a survey program, to gain information about the spatial distribution of the stock in order to optimize the survey. A pilot study for the annual fishery-independent survey program for the green sea urchin fishery was initialized in Maine in the summer of 2001. The pilot study was extensive, time-consuming and costly, and needed to be optimized to ensure its feasibility as a long-term scientific survey. The high degree of spatial variability in sea urchin abundance, however, prevented us from using standard optimization techniques, such as traditional statistics or even geostatistics. Kernel estimation and computer simulations were used to characterize the large-scale spatial density structure of the sea urchin population and investigate how different sampling strategies effected realizations of the density structure. Since realizations of the large-scale density structure are the vital components of the sea urchin stock assessment (Chapter 3), any changes in this structure would dramatically alter the outcome of the assessment. Therefore, we defined an optimal sampling strategy as a design that produces realizations of the large-scale spatial structure that are similar to the original population while using less sampling intensity than the original sampling strategy. Using the original survey strategy, a reduction of sampling intensity to 10 sites per strata, or 90 total sites, was optimal because it corresponded to a large decrease in effort but only a marginal decrease in precision. However, a regular sampling strategy, with approximately 90 grids arranged along the coastline, provided the highest precision for the green sea urchin fishery

when analyzed solely with spatial statistics. Considering that the sea urchin data will be analyzed by traditional and spatial statistics, we believe that the original stratified random sampling design reduced to 10 locations per strata is the most sensible optimization for the Maine green sea urchin fishery-independent survey program at this time.

Introduction

Fishery-independent surveys are scientific studies designed to provide biological and ecological information on a fish stock (Hilborn and Walters 1992; Jennings 2001). They can generate high quality data, with small variances and biases, which are representative of the entire targeted fish population. Stock assessments based on fishery-independent data have less uncertainty and bias than ones based on fishery-dependent data, which are generated from normal fishing activities. Therefore, fishery-independent surveys are essential for stock assessments and for developing appropriate management plans (Hilborn and Walters 1992). To establish an effective fishery-independent survey program, pilot studies should be conducted prior to the start of the survey program in order to gain information about the spatial distribution of the stock and to identify environmental variables that influence this distribution. The pilot study is then redesigned, or optimized, based on the information collected and on the future analysis plan (Andrew and Mapstone 1987; Kitsiou et al. 2001). According to Andrew and Mapstone (1987), "Optimization of the design of sampling programmes is achieved by determining the most efficient allocation of resources-*i.e.*, minimizing decreases in precision and/or resolution imposed by cost or by logistical constraints."

A pilot study for an annual fishery-independent survey program was initialized in the summer of 2001 for the green sea urchin fishery in Maine. The pilot study was designed and implemented to provide detailed information on the population structure, spatial variability and biological/ecological characteristics of the sea urchin stock along the coast of Maine. The pilot study was extensive, time-consuming and costly, and could not be maintained for the annual survey. Therefore, the pilot study needs to be optimized to reduce the cost while maintaining high precision and accuracy of the annual survey.

Many statistical techniques have been developed to optimize sampling programs, including traditional experimental design, geostatistics and Monte Carlo computer simulation (Cochran 1977; Rivoirard 2000; Petitgas 2001). Traditional statistical methods are primarily based on random sampling and optimization usually involves stratification of the study area based on the spatial structure of the stock (Cochran 1977; Hilborn and Walters 1992). The study area is divided into smaller regions, or strata, using variables that influence the spatial structure of the stock, such as depth or habitat, in order to increase sampling precision and accuracy. Optimization with traditional statistics is limited, though, because these methods assume that the fish stock is distributed randomly over the study area or strata. Truly random distribution in a fished stock is rare, however, most stocks exhibit spatial patterns or dependence, also known as spatial heterogeneity. A different branch of statistics, known as spatial statistics, is specifically designed to investigate the spatial distribution of a stock and can be used for survey design optimization.

Spatial statistics or spatial analyses are employed to model first and second-order, or large and small-scale, spatial variability of a variable, such as fish abundance, in order to estimate the value at unobserved locations (Bailey and Gatrell 1995; Petitgas 2001). Intrinsic second-order methods have become the most popular geostatistical tools and the kriging variance, or mean square prediction error, can been used to compare survey designs for optimizing fishery surveys (Pelletier and Parma 1994; Rivoirard et al. 2000; van Groenigen 2000; Petitgas 2001). Two assumptions must be met in order to use intrinsic geostatistical methods. First, the spatial distribution of the stock cannot be affected by the geometry of the region, *i.e*. the spatial distribution cannot differ near the borders of the zone (Petitgas 1993; Bailey and Gatrell 1995; Warren 1998; Rivoirard et al 2000). Second, the process must exhibit some degree of second-order stationarity, or spatial dependence, which means that small-scale deviations in variables are similar in neighboring sites. In chapter 3, the suitability of the green sea urchin data for analysis with intrinsic geostatistics was addressed. The data did not satisfy the assumptions, especially for stationarity; the sea urchin data are too highly skewed and spatially variable. Since the assumptions are violated, we must use other spatial analysis techniques to characterize the spatial variability of the stock (Bailey and Gatrell 1995; Warren 1998; Petitgas 2001). .

Several spatial analysis techniques are available for investigating the large-scale variations in fish stock abundance (Bailey and Gatrell 1995). For example, in Chapter 3, triangulated irregular networks (TINs) were used to estimate exploitable biomass for the green sea urchin fishery. TINs are good estimators of large-scale spatial patterns but require relatively evenly spaced sampling locations; its performance decreases when sampling locations become clustered (ESRI 1998; Guan et al 1999). Kernel estimation is an advanced form of weighted spatial moving averages that can be used with any type of sampling strategy: random, clustered or grids (Bailey and Gatrell 1995). It does not require any major assumptions nor does it require complex statistical decisions or modeling. Therefore, kernel estimation is used to estimate the large-scale patterns in sea urchin stock abundance, but since it does not incorporate a variance structure, it cannot be directly used for sample design optimization.

Kernel estimation paired with computer simulations may provide the framework necessary for optimizing survey programs. Computer simulation approaches have been increasingly used in fisheries due to their ability to incorporate different sources of variations, especially spatial and temporal heterogeneity (e.g. Hilborn and Walters 1987; Horppila and Peltonen 1992; Andrew and Chen 1997). A simulation approach allows researchers to investigate how uncertainty in the spatial structure of a fished stock can affect survey programs and stock assessments. Sampling programs based on random sampling theory can have countless realizations, and the precision of one realization may not represent the precision of the sampling program. Simulations allow us to produce multiple realizations and estimate the mean precision of a sampling strategy.

The objective of this project is to develop a framework that incorporates spatial statistics and computer simulations to identify an optimum sampling strategy. An optimal sampling strategy should provide the most accurate and precise information on a stock, as possible. Since we are using spatial statistics, we are most interested in the large-scale spatial structure of sea urchin density. The combination of kernel estimation and computer simulation allows us to estimate the large-scale spatial density structure and determine how different sampling strategies effect realizations of this structure. Since these realizations are the vital components of the sea urchin stock assessment , any changes in these structures would dramatically alter the outcome of the assessment. Therefore, we define an optimal sampling strategy as a design that produces realizations of the large-scale spatial structure that are similar to the original population while using less sampling intensity than the original sampling strategy.

Materials and Methods

Urchin density and size frequency information were obtained from the 2001 pilot study for the State's annual fishery independent survey. The Department of Marine Resources employed a stratified random sampling design, where 16 sites were sampled in each of 9 strata along the Maine coast exclusively in potential urchin habitat (rock or gravel substrate) (Figure 1). To minimize the sample variances within the strata, the width of each stratum was inversely proportional to the commercial landings in the region. At each site, 90 quadrats ($1m^2$) were randomly sampled along a linear transect set perpendicular to shore using SCUBA. All urchins within the quadrat were counted and test diameter was measured. Sampling intensity was equally divided over three depth zones: 0-5m, 5-10m, and 10-15m. Mean site densities were calculated, as were mean site densities by depth zones to allow each depth stratum to be analyzed separately.

A simulation framework was developed to test the ability of different sampling programs to recreate the large-scale spatial structure of the sea urchin population (Figure 4.2.). Mean sea urchin densities by depth zone, as well as bathymetry and suitable urchin substrate data, were the initial inputs for the framework. Kernel estimation was used to estimate the large-scale variations in the green sea urchin stock by depth zone (Bailey and Gatrell 1995). The kernel estimate for mean urchin density at a location is calculated as

$$\hat{\mu}_\tau = \frac{\sum_{i=1}^{n} k(\frac{s - s_i}{\tau}) y_i}{\sum_{i=1}^{n} k(\frac{s - s_i}{\tau})}, \tag{1}$$

where $\hat{\mu}_\tau$ is the mean urchin density; k is the kernel, or bivariate probability function; **s** is the location (x,y) where the urchin density is being estimated; $\mathbf{s}_i$ are the locations where the urchin densities were sampled; τ is the bandwidth, or the radius of the moving window;

and y_i is the urchin density. The study area was converted into an ASCII raster image (1500 x 1178 pixels, pixel=236.93 m) and weighted averages were computed for every pixel based on a quartic kernel. A bandwidth, in pixels, was selected to minimize error and ensure adequate coverage and smoothness. The kernel estimation technique produced plots of smoothed urchin densities by depth zone. These plots were modified to only include areas of the rock/gravel substrate, in effect producing spatial representations of the population density structure (Figure 4.3). These original density plots were used to test different sampling strategies and gauge their relative performance. The sampling strategies varied based on the number of sites and number and size of strata, allowing us to test the following survey designs: random, stratified random with equal strata width, and stratified random with strata based on the original survey design (Table 4.1.). These sampling designs were chosen because they were feasible for the program and are routinely used in fishery surveys. Resampling was conducted randomly within the potential urchin habitat in the appropriate depth zone, producing sets of urchin densities by location. New urchin density plots were created from these observations using the kernel estimation technique. The number of simulations was limited to 50 due to restraints placed on computing power imposed by the large size of the files.

The performance of a sampling strategy was evaluated using mean squared error (MSE). The MSE has been used to determine optimal sampling strategies for fisheries and is calculated as

$$(2) \qquad MSE(Q) = \frac{\sum_{N=1}^{N}(Q_S - Q_O)^2}{N},$$

where Q_O is the stock density value from the original density plot, Q_S is the stock density value from the sampled plot, and N is the number of simulations (Cochran 1977; Guan et al. 1999). MSE was calculated for each pixel in the urchin density plots (n=1,767,000), creating a plot of MSE for each sampling strategy. A mean MSE value was calculated for each plot from the pixel MSE values to facilitate selection of an optimal sampling strategy. An optimal sampling strategy is a design that minimizes mean MSE while using less sampling intensity than the original pilot study.

Results

The first kernel estimation step produced the original density plots, which characterizes the large-scale spatial variations in the sea urchin stock (Figure 4.3.). After implementing a sampling strategy, the second kernel estimation step created the sampled density plots (Figure 4.4.). Finally, plots of MSE were created for each scenario by calculating MSE per pixel (Figure 4.5.).

The stratified random strategy from the pilot study was tested using the 3 depth zone datasets and an average site dataset. MSE values for depth zone 1, 0-5 m, were considerably higher than the other datasets (Figure 4.6.). This result suggested that depth zone 1 had the highest spatial variability, so recreations of the large-scale variations in urchin density were the least precise. This dataset was used in all subsequent analyses because it is the most variable urchin density structure; it represents a worst-case scenario. A reduction of sampling intensity to 10 sites per strata, or 90 total sites, corresponded to a large decrease in effort but only a marginal decrease in precision (Figure 4.6.). MSE at a sampling intensity of 90 sites was used as a reference point for comparison between sampling strategies.

None of the tested sampling strategies had consistently lower MSE values than the original pilot study design, over all sampling intensities (Figure 4.7.). At low sampling intensities (less than 27 sites) random sampling had the lowest MSE. At greater than 27 sites, the original survey had the lowest MSE values over the majority of sampling locations. However, when sampling intensity was set at 90 sites, MSE values for the stratified random strategies with equal strata width dropped below the original survey design at higher levels of stratification.

At 90 sites, sampling strategies with greater than 9 equal sized strata had lower MSEs than the original survey design (Figure 4.8.). MSE values decreased with increasing stratification, reaching a minimum at 45 strata with 2 sampling locations. The original survey strategy performed better, with 90 sites, than random sampling (1 strata) and all stratified random strategies with less than 9 equal sized strata.

Discussion

In optimization studies, we assume that the population was oversampled so the data collected is representative of the entire population. We believe that this assumption is valid for the 2001 pilot study for the green sea urchin fishery-independent survey; therefore we can legitimately optimize the survey. We defined an optimal sampling strategy as a design that produces realizations of the large-scale spatial structure that are similar to the original population while using less sampling intensity than the original sampling strategy. Within the original survey design, MSE quickly decreased and leveled off as sampling intensity increased (Figure 4.6.). We chose 90 sites as a reference point for this sampling strategy because it corresponded to a large decrease in sampling effort, a marginal increase in MSE and was buffered from the high MSE values at lower sampling intensities. When comparing amongst other sampling designs, however, the original sampling strategy did not have the lowest MSE.

In our study, the stratified random strategy with equal strata width had comparable or higher precisions than the original stratified random strategy. In particular, sampling strategies with more than 9 equal sized strata had considerably lower MSE values than the original sampling strategy (Figure 4.8.). Interestingly, MSE decreased further with added stratification. The high levels of stratification most likely caused this increase in precision. As the number of strata increased, and correspondingly the number of sampling locations per strata decreased, the sampling strategy more closely resembled a regular, or grid, sampling strategy. Grids have long been considered ideal sampling strategies for analysis with spatial statistics (Haining 1990, Rivoirard et al. 2000; Petitgas 2001). In fact, as long as the spatial process is not periodic, grids are the preferred option (Haining 1990; Simard et al. 1992). Accordingly, a regular sampling strategy, with 90 grids arranged along the coastline, would provide the highest precision for the green sea urchin fishery when analyzed with spatial statistics.

Currently the green sea urchin fishery is not analyzed solely with spatial statistics, though. Fishable biomass was estimated with spatial statistics (Chapter 3), while stock assessments (Chen and Hunter 2003) and investigations into biological reference points (Chapter 1) have been conducted using fisheries population dynamics and computer simulation techniques. Therefore, the optimal sampling strategy not only needs to satisfy the original criteria, *i.e.* minimizing decreases in precision while reducing sampling intensity, but, additionally, must be suitable to the future analysis plans (Andrew and Mapstone 1987). A regular sampling strategy may be the preferred design for analyzing the sea urchin stock with spatial statistics but it is not preferred for traditional statistics. When used with traditional statistics, regular sampling strategies can yield greater precision, but estimates are usually biased and sample variance cannot be directly estimated from the samples (Cochran 1977: Hilborn and Walters 1992). Conversely, stratified random sampling strategies are appropriate for both traditional statistics and spatial statistics. In traditional statistics, stratified random strategies have greater precision than random designs if the variance of a variable per strata is less than the overall variance (Hilborn and Walters 1992). In spatial statistics, stratified random strategies can have lower variances than random and grid designs, especially if there is a spatial trend (Haining 1990). So, a careful designed stratified random strategy, where strata size reduces sampling variance, would be more flexible for analysis than a regular sampling strategy.

An optimal sampling strategy must balance many factors, ranging from logistics and cost, to precision and analysis techniques. We believe that the original stratified random sampling strategy with reduced sites per strata is the best compromise and a sensible optimization for the Maine green sea urchin fishery-independent survey program at this time. Further simulation work on optimization should continue in order to investigate different sampling designs using more simulations.

Acknowledgement

We would like to thank the support by the Northeast Consortium, Maine Department of Marine Resources, and Maine Sea Urchin Zone Council. Discussions with Dr. Robert Vadas, Dr. Larry Harris, and Dr. Linda Mercer are very helpful.

REFERENCES

Andrew NL, Chen Y. 1997. Optimal sampling for estimating the size structure and mean size of abalone caught in a New South Wales fishery. Fishery Bulletin 95:403-413.

Andrew NL, Mapstone BD. 1987. Sampling and the description of spatial pattern in marine ecology. Barnes M, editor. Aberdeen, U.K.: Aberdeen University Press. 39-90 p.

Bailey T, Gatrell A. 1995. Interactive Spatial Data Analysis. Essex, England: Pearson Education. 413 p.

Chen Y, Hunter M. 2003. Assessing the green sea urchin (Strongylocentrotus droebachiensis) stock in Maine, USA. Fisheries Research 60:527-537

Cochran, WG. 1977. Sampling Techniques. New York: John Wiley. 428 p.

Environmental Systems Research Institute, Inc. (ESRI). 1998. ARC/INFO Version 7.2.1. Redlands, CA.

Guan W, Chamberlain RH, Sabol BM, Doering PH. 1999. Mapping Submerged Aquatic Vegetation with GIS in the Caloosahatchee Estuary: Evaluation of Different Interpolation Methods. Marine Geodesy. 22(2):69-92.

Haining R. 1990. Spatial Data Analysis in the Social and Environmental Sciences. Cambridge: Cambridge University Press.

Hilborn R, Walters C. 1987. A general model for simulation of stock and fleet dynamics in spatially heterogeneous fisheries. Canadian Journal of Fisheries and Aquatic Sciences 44:1366-1369.

Hilborn R, Walters CJ. 1992. Quantitative fisheries stock assessment: Choice, dynamics and uncertainty. New York: Chapman and Hall. 570 p.

Horppila J, Peltonen H. 1992. Optimizing sampling from trawl catches: contemporaneous multistage sampling for age and length structures. Canadian Journal of Fisheries and Aquatic Sciences 49:1555-1559.

Jennings S, Kaiser M, Reynolds JD. 2001. Marine Fisheries Ecology. Oxford: Blackwell Science. 417p.

Kelley JT, Barnhardt WA, Belknap DF, Dickson SM, Kelley AR. 1999. The seafloor revealed. Maine Geological Survey. 55 p.

Kitisiou D, Tsirtsis G, Karydis M. 2001. Developing an optimal sampling design: A case study in coastal marine ecosystem. Environmental Monitoring and Assessment. 71:1-12.

Pelletier D, Parma AM. 1994. Spatial distribution of Pacific halibut (*Hippoglossus stenolepis*): An application of geostatistics to longline survey data. Canadian Journal of Fisheries and Aquatic Sciences 51(7):1506-1518.

Petitgas P. 1993. Geostatistics for fish stock assessments: A review and an acoustic application. ICES Journal of Marine Science 50(3):285-298.

Petitgas P. 2001. Geostatistics in fisheries survey design and stock assessment: Models, variances and applications. Fish & Fisheries Series 2(3):231-249.

Rivoirard J, Simmonds J, Foote KG, Fernandes P, Bez N. 2000. Geostatistics for estimating fish abundance. Oxford: Blackwell Science. 206 p.

Simard Y, Legendre P, Lavoie G, Marcotte D. 1992. Mapping, estimating biomass, and optimizing sampling programs for spatially autocorrelated data: Case study of the northern shrimp (*Pandalus borealis*). Canadian Journal of Fisheries and Aquatic Sciences 49(1):32-45.

van Groenigen JW. 2000. The influence of variogram parameters on optimal sampling schemes for mapping by kriging. Geoderma 97:223-236

Warren WG. 1998. Spatial analysis for marine populations: factors to be considered. In: Jamieson GS, Campbell A, editors. Proceedings of the North Pacific Symposium on Invertebrate Stock Assessment and Management: Canadian Special Publication of Fisheries and Aquatic Science p 21-28.

Table 4.1. Summary of the sampling strategies evaluated in this study.

Sampling strategy	Number of strata	Sites per strata
Original Stratified Random Strata width dependent on landings	9	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
Random	1	10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150
Stratified Random Equal strata width	2	6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72
	3	4, 8, 12, 16, 20, 24, 28, 32, 36. 40, 44, 48
	4	3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36
	5	3, 6, 9, 12, 15, 18, 21, 24, 27, 30
	6	2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24
	7	2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22
	8	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18

Table 4.1. Contd

For the stratified random strategies with equal strata width, strata are defined as equal subdivisions of the coast along an east-west axis.

St		,
Equal strata width		12, 13, 14, 15, 16
	10	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
	11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
	12	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
	15	6
	18	5
	30	3
	45	2
	90	1

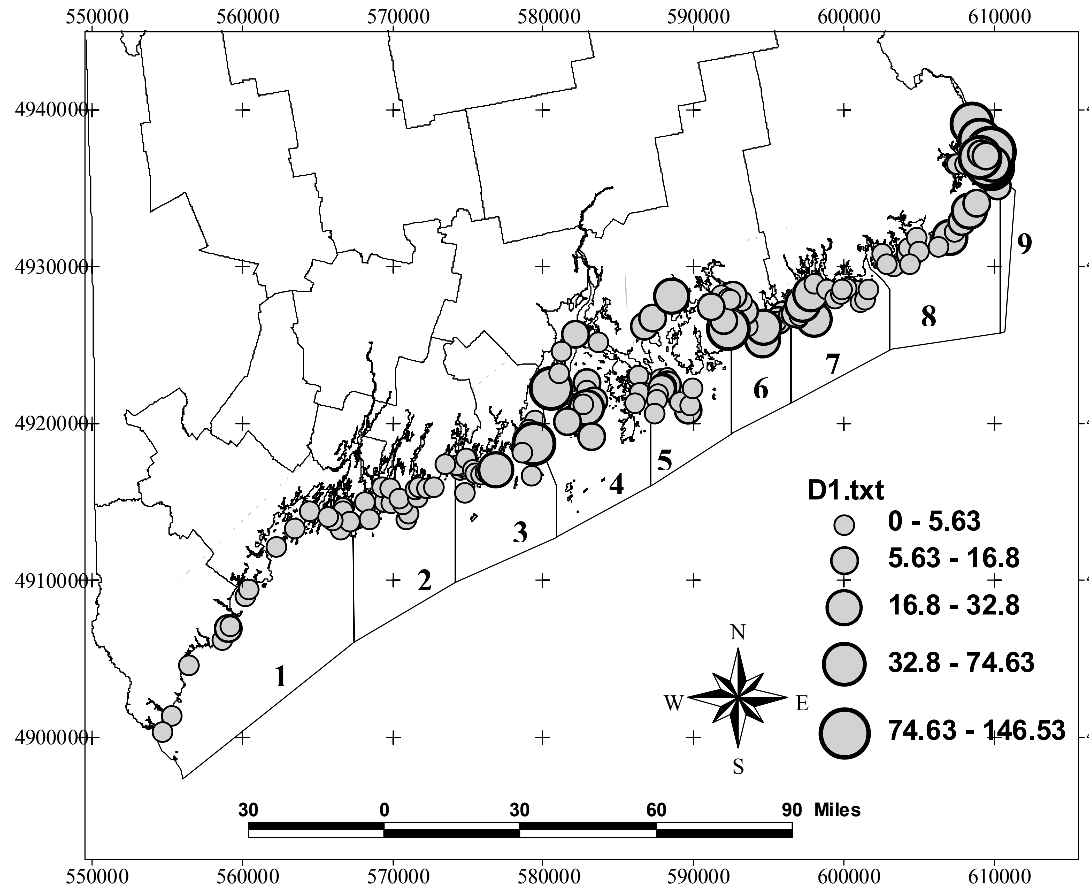

Figure 4.1. Mean urchin densities (urchins m^{-2}) for the 0-5 m depth zone from the 2001 pilot study. Strata from the original stratified random strategy are labeled.

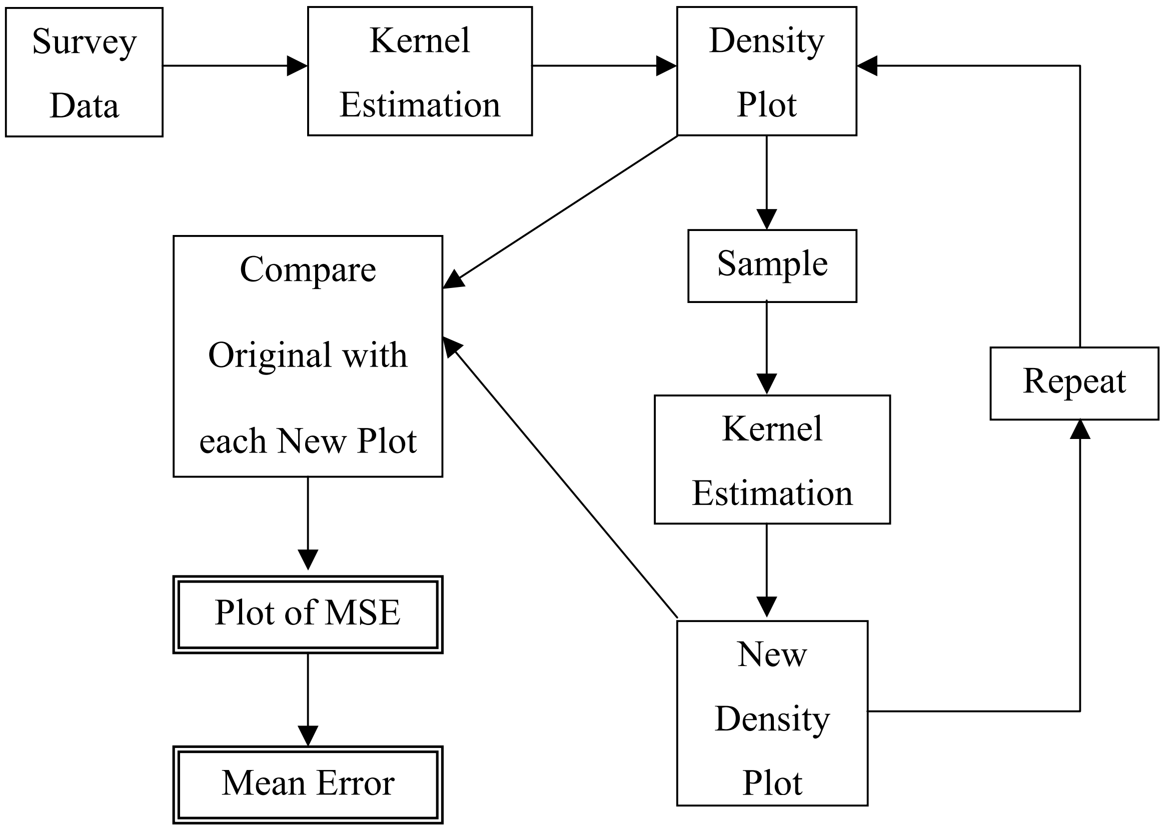

Figure 4.2. Flowchart of simulation approach to estimate the variance associated with a sampling strategy for the Maine sea urchin fishery.

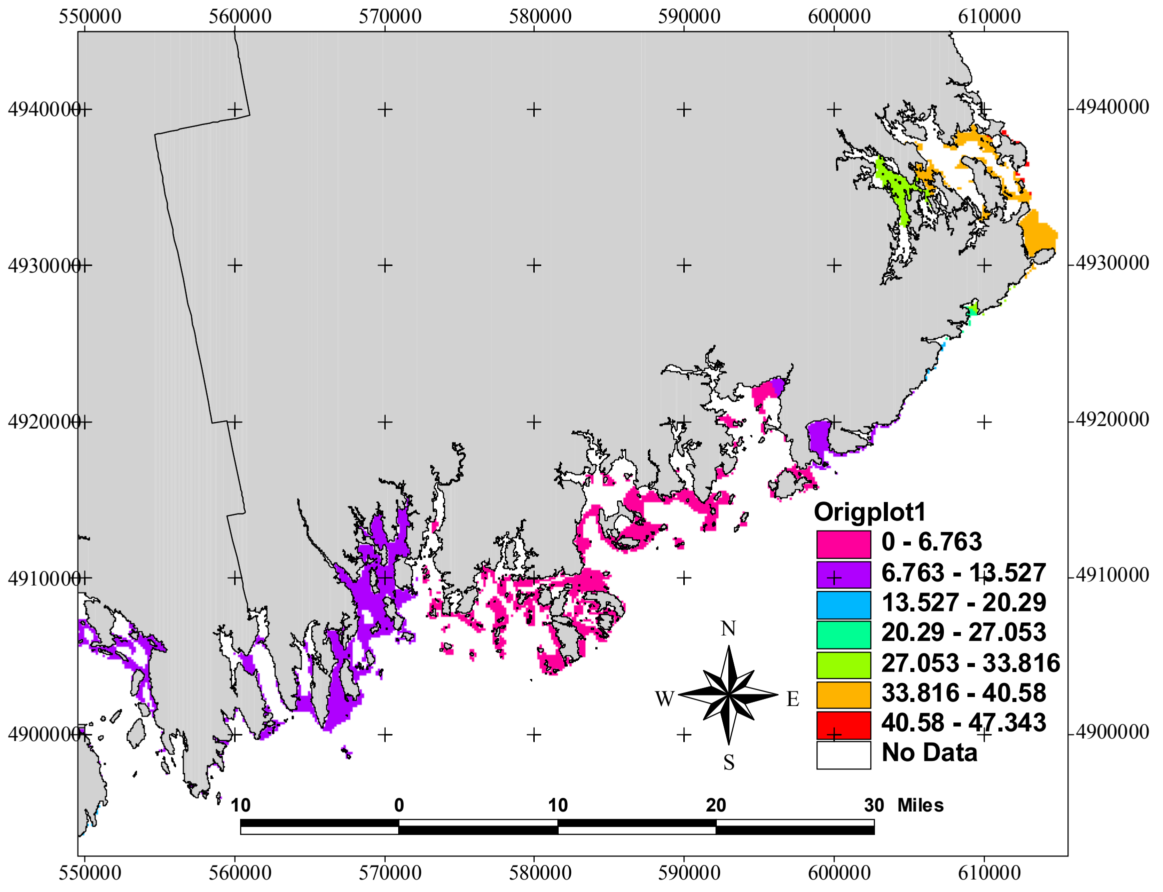

Figure 4.3. Original density plot characterizing the large-scale spatial variations in stock for density (urchins m^{-2}) depth zone 1 (0-5m) in areas west of Mt. Desert Island.

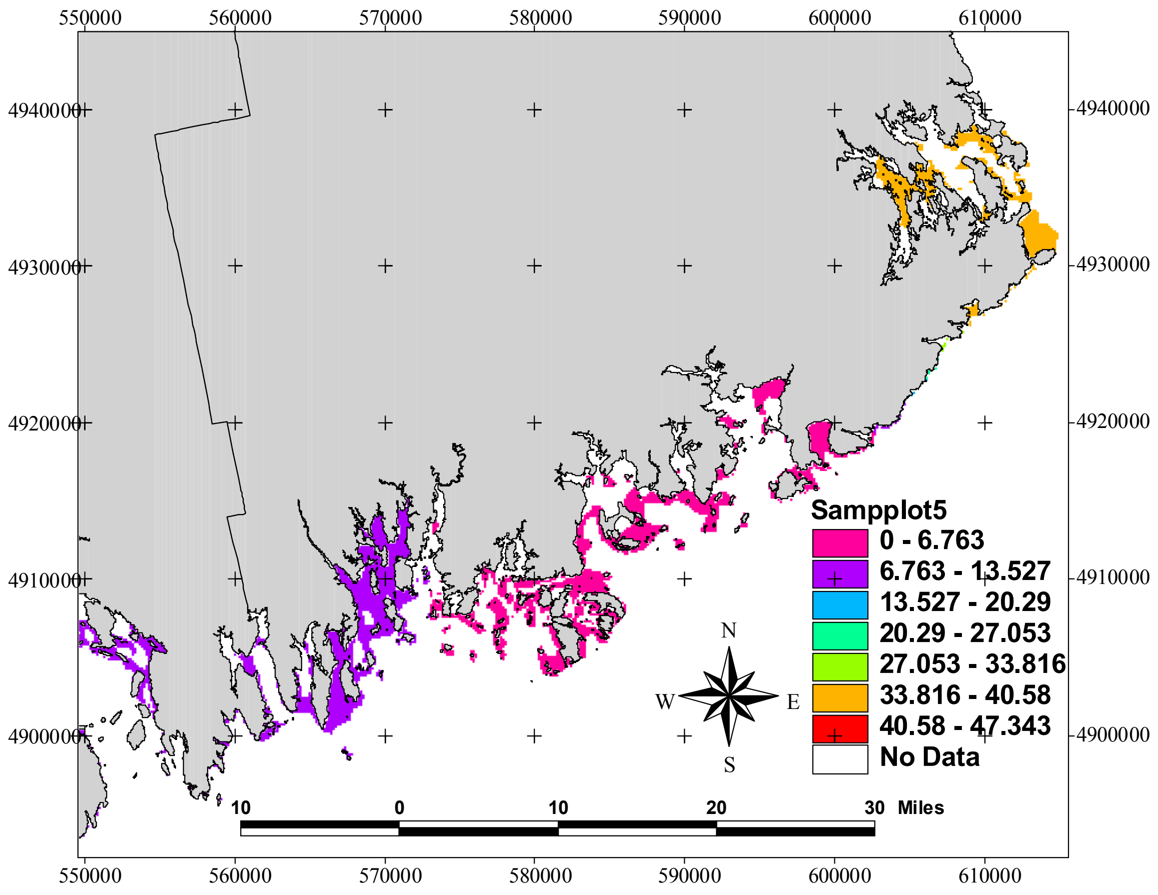

Figure 4.4. One simulation of a sample density plot (urchins m^{-2}) created by sampling the original density plot with the original stratified random design using 10 sites per strata in areas west of Mt. Desert Island.

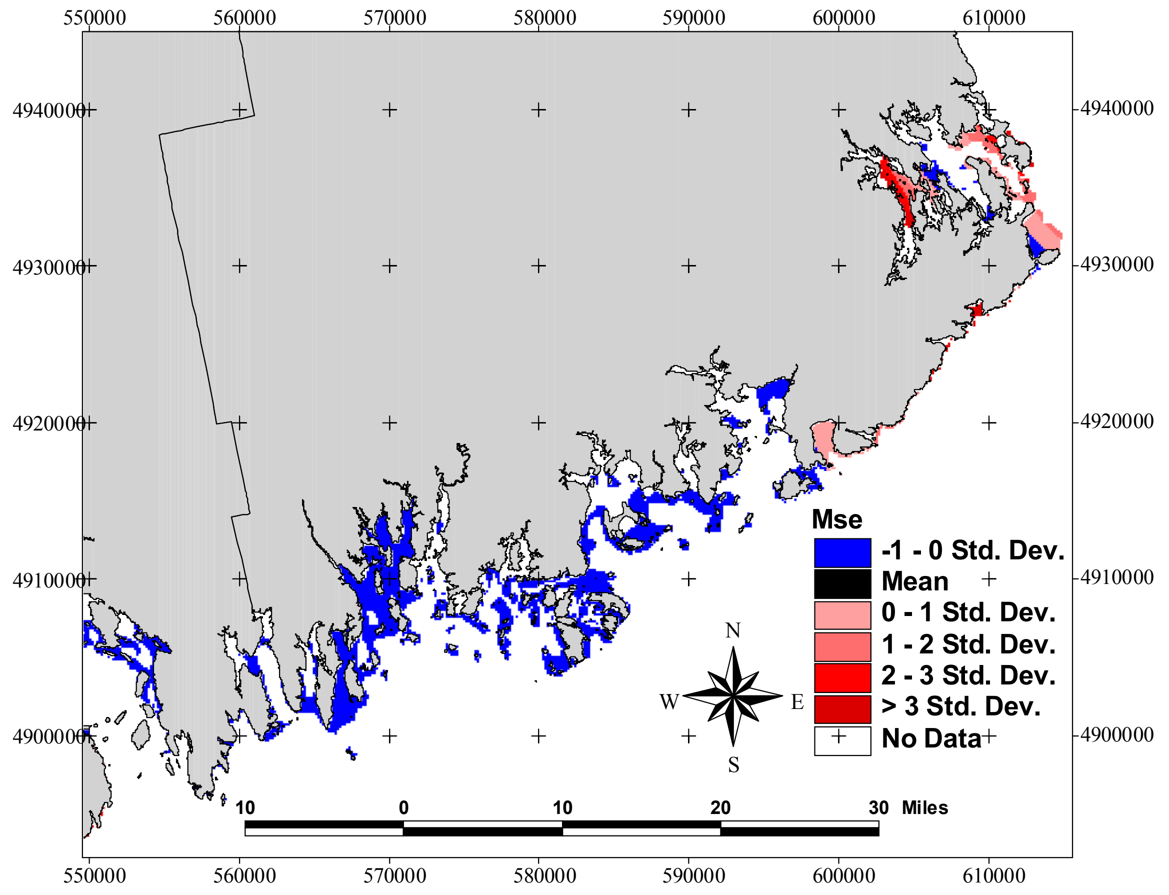

Figure 4.5. Plot of MSE for the original stratified random design using 10 sites per strata in areas west of Mt. Desert Island. Mean MSE is 2.90.

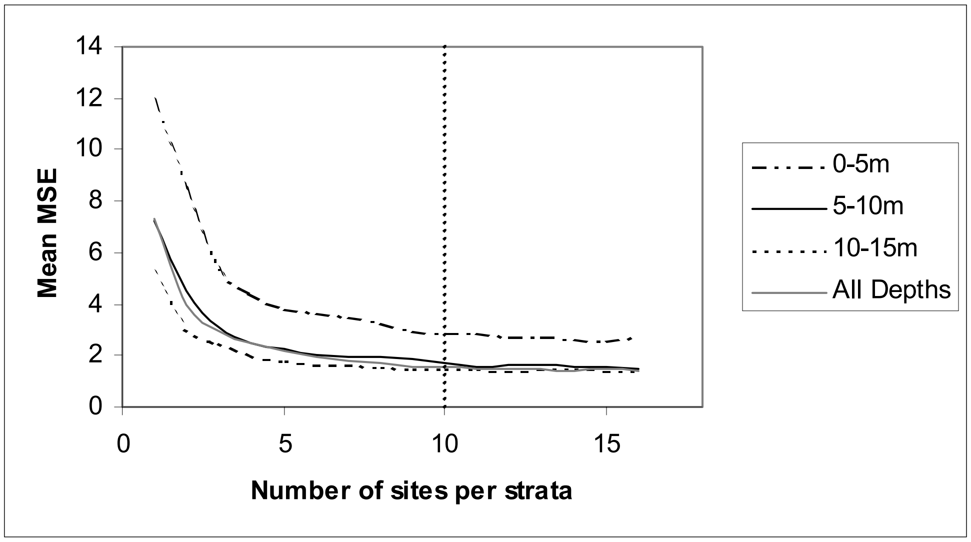

Figure 4.6. Mean squared error (MSE) as a function of the number of sites sampled per strata, using the original survey design by depth zone. The dashed line represents 90 sampling locations, 10 sites in each of 9 strata, which was used for comparisons amongst different sampling strategies.

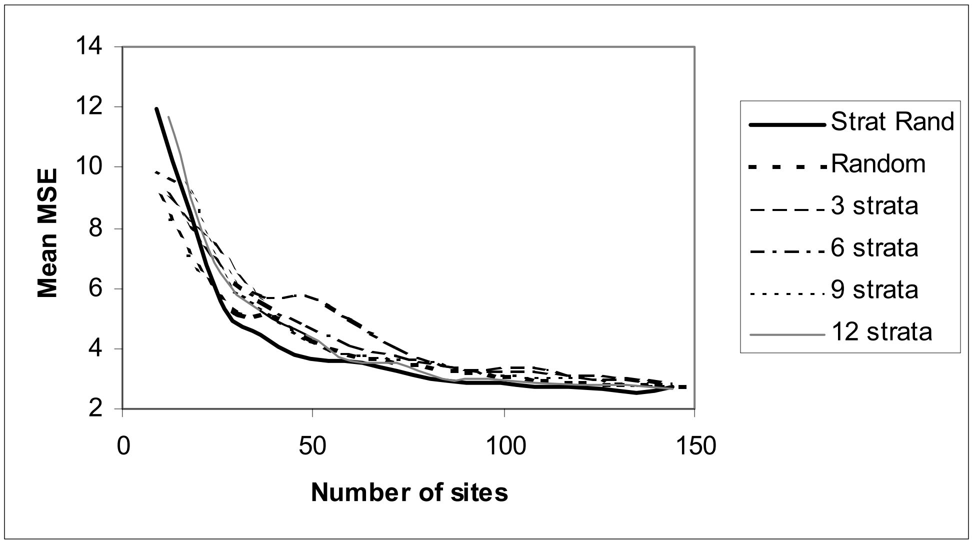

Figure 4.7. Mean squared error (MSE) as a function of the number of sites for the original stratified random sampling strategy (Strat Rand), random sampling, and stratified random sampling with equal strata width (3-12 strata) for depth zone 1 using the simulation framework approach.

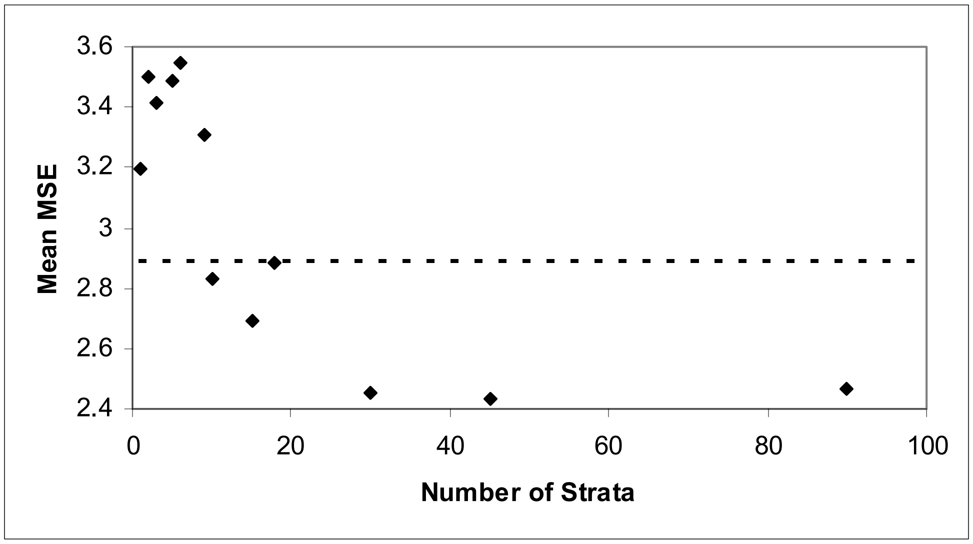

Figure 4.8. Mean squared error (MSE) for stratified random sampling strategies with equal strata width using 90 samples. The dashed line represents the MSE for the original sampling strategy with 90 sites, 10 in each of the 9 unequally sized strata.

Appendix

PROCEDURE AND COMPUTER CODE FOR IDENTIFYING OPTIMAL SAMPLING STRATEGIES

Procedure for identifying optimal sampling strategies

1. Create a text file of fish densities by location. Place the x coordinate in the first column, the y coordinate in the second column and the density value in the third column. Do not include column headings in the text file.
2. Create an ArcASCII template file. This file indicates what regions have suitable habitat and potential fish abundance. The sampling program will be limited to these regions. Note: a buffer zone should be created around the region of interest in valid.asc. The width of the buffer zone should be equal to or greater than the size of the moving window (kernel).
    a. The file can be created directly in an ASCII format or it can be converted from other spatial formats, such as shapefiles, TINs, and grids, using the ArcToolbox program from ArcInfo 7.1.
3. Rename the urchin density text file "obs.txt" and the template ASCII to "valid.asc." and the bathymetry ASCII to "gridC.asc." Place these files in the same folder as the C++ kernel estimation program.
4. Run the C++ kernel estimation program. Follow the directions on the program. Note: The C++ code is designed for a stratified random strategy with a set number and size for the strata. The "Size of the moving window" is the kernel length and is

equivalent to the radius of a circle in pixels. We recommend limiting the number of simulations because the ArcASCII files can be very large.

5. When the program terminates, enter 1. Then run the C++ mean squared error (MSE) estimation program. The program will output the mean MSE and create an ASCII file of MSE.

<u>C++ computer code for kernel estimation and implementation of a stratified random sampling strategy</u>

```
// biomass.cpp : calculates biomass of an arc ascii grid A based on constraints
//
        definde by an arc ascii grid B.


#include "stdafx.h"
#include "biomass.h"
#include <fstream.h>
#include "math.h"
#include "Matrix.h"
#include "Location.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


        // Macro to get a random integer with a specified range
        #define getrandom(min, max) \
                ((rand()%(int)(((max) + 1)-(min)))+ (min))



CString int_to_string(int number, CString startstring)
{
        bool done = false;

        while (!done)
        {
```

```
		if ((number/10) < 1)
		{
			if (number == 0)
				startstring = "0" + startstring;
			if (number == 1)
				startstring = "1" + startstring;
			if (number == 2)
				startstring = "2" + startstring;
			if (number == 3)
				startstring = "3" + startstring;
			if (number == 4)
				startstring = "4" + startstring;
			if (number == 5)
				startstring = "5" + startstring;
			if (number == 6)
				startstring = "6" + startstring;
			if (number == 7)
				startstring = "7" + startstring;
			if (number == 8)
				startstring = "8" + startstring;
			if (number == 9)
				startstring = "9" + startstring;

			done = true;
		}

		if ((number/10) >= 1)
		{
			startstring = int_to_string((number-(int(number/10)*10)),
startstring);
			number = int(number/10);
		}
	}

	return startstring;
}


/////////////////////////////////////////////////////////////////////////////
// The one and only application object

CWinApp theApp;

//using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
```

```
{
	int nRetCode = 0;

	// initialize MFC and print and error on failure
	if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
	{
		// TODO: change error code to suit your needs
		cerr << _T("Fatal Error: MFC initialization failed") << endl;
		return nRetCode = 1;
	}
//////////////////////////////////////////////////////////////////////////////
//////// MY CODE


	ifstream inFile;  // Input data file.
	ofstream outFile; // Output data file.
	CMatrix gridA, mastergridA, tempgrid, validgrid;
	char chardummy;
	int ncols, nrows, urxwin, urywin, xsample, ysample;
	int intdummy = 0, i, j, k, nodata=-9999, n, max=0, min=0, maxout, runs=0;
	double res, doubledummy, urchins, tau, kf, d;
	bool data;
	CLocation sample, tempobs;
	CList<CLocation,CLocation&> observations;
	CString filename, filename2;

	double llx, lly, llxwin, llywin, count;
	double winllx, winlly, winurx, winury, winarea;
	int windowsize, sampleloop;
	double wincenterx, wincentery;

	const pi=3.141592653589793;


	cout << "Enter 1 to load observations (obs.txt):";
	cin >> intdummy;
	cout << "\n";


	gridA.Empty();
	tempgrid.Empty();
	validgrid.Empty();
	mastergridA.Empty();


	inFile.open("obs.txt");
	if(!inFile)

	{
		cout << "Error opening file\n";
		return nRetCode;
	}

	while(inFile)
	{
		inFile >> sample.x >> sample.y >> sample.urchincount;
		if (inFile) observations.AddTail(sample);
	}

	inFile.close();


	cout << "Enter 1 to load the valid.asc grid outline (will be used for llx, lly, and 
resolution): \n";
	cin >> intdummy;
	cout << " \n";

	inFile.open("valid.asc");
	if(!inFile)

	{
		cout << "Error opening file\n";
		return nRetCode;
	}

	inFile >> chardummy >> chardummy >> chardummy >> chardummy >> 
chardummy;
	inFile >> ncols;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >> 
chardummy;
	inFile >> nrows;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >> 
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> llx;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >> 
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> lly;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >> 
chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> res;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy;
	inFile >> nodata;

	validgrid.SetMatrixSize(CSize (ncols, nrows));


	for (i=0;i<nrows;i++)
		{
			for (j=0;j<ncols;j++)
			{
				inFile >> doubledummy;
				validgrid.SetAt(CPoint (j,i), doubledummy);
			}
		}

	inFile.close();


	gridA.SetMatrixSize(CSize (ncols,nrows));
	tempgrid.SetMatrixSize(CSize (ncols,nrows));
	mastergridA.SetMatrixSize(CSize (ncols,nrows));

	cout << "Please enter the size of the moving window (half the side in pixel): \n";
	cin >> windowsize;
	cout << "\n";

	tau = (windowsize*res) + (res*0.5);

	winarea = ((2*windowsize*res + res) * (2*windowsize*res + res));


	for (i=0;i<nrows;i++)
	{
		for (j=0;j<ncols;j++)
		{
			gridA.SetAt(CPoint (j,i),nodata);
		}
	}


	POSITION pos = observations.GetHeadPosition();

	for (i=windowsize;i<(nrows-windowsize);i++)
	{
		for (j=windowsize;j<(ncols-windowsize);j++)
		{

			urchins = 0;
			count = 0;

			wincenterx = llx + j*res + 0.5*res;
			wincentery = lly + nrows*res - (i*res + 0.5*res);
			winllx = wincenterx - (0.5*res + windowsize*res);
			winlly = wincentery - (0.5*res + windowsize*res);
			winurx = wincenterx + (0.5*res + windowsize*res);
			winury = wincentery + (0.5*res + windowsize*res);

			pos = observations.GetHeadPosition();

			for (k=0;k<observations.GetCount();k++)
			{
				tempobs = observations.GetNext(pos);

				if ((tempobs.x >= winllx) && (tempobs.x < winurx) &&
(tempobs.y >= winlly) && (tempobs.y < winury))
				{
					d = sqrt(((wincenterx - tempobs.x)*(wincenterx -
tempobs.x)) + ((wincentery - tempobs.y)*(wincentery - tempobs.y)));

					if (d <= tau)
					{
						kf = (3/pi)*((1-((d/tau)*(d/tau)))*(1-
((d/tau)*(d/tau))));

						urchins = urchins + (kf *
tempobs.urchincount);
						count = count + kf;


					}
				}
			}

			if ((validgrid.GetAt(CPoint (j,i)) != nodata))
			{
				if (count == 0)
					gridA.SetAt(CPoint (j,i),nodata);

				else
					gridA.SetAt(CPoint (j,i),(urchins/count));
```

```
                }

        }
}



cout << "Number of Samples per Strata: ";
cin >> n;
cout << "\n";

cout << "Number of runs is set to 20";
runs = 20;
cout << "\n";

for (i=0;i<nrows;i++)
{
        for (j=0;j<ncols;j++)
        {
                mastergridA.SetAt(CPoint (j,i), (gridA.GetAt(CPoint (j,i))));
        }
}


for (sampleloop=0;sampleloop<runs;sampleloop++)
{
        filename = "";
        filename = int_to_string((sampleloop+1),filename);
        filename = "sample_set" + filename + ".txt";


        for (i=0;i<nrows;i++)
        {
                for (j=0;j<ncols;j++)
                {
                        gridA.SetAt(CPoint (j,i), (mastergridA.GetAt(CPoint (j,i))));
                }
        }


        outFile.open(filename);
        if(!outFile)
        {
                cout << "Error opening file\n";
                return nRetCode;
        }
```

```
		//Zone 1

		llxwin = int((362383.06-llx)/res);
		llywin = (nrows - int((4768863.7-lly)/res));
		urxwin = int((431450.46-llx)/res);
		urywin = (nrows - int((4866671.22-lly)/res));

		for (i=0;i<n;i++)
		{
			data = false;
			maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

			while (!data)
			{

				xsample = getrandom(llxwin,urxwin);
				ysample = getrandom(urywin,llywin);


				if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
				{
					outFile << (xsample*res+llx) << "\t" << ((nrows - 
ysample)*res+lly) << "\t";
					outFile << gridA.GetAt(CPoint (xsample,ysample)) 
<< "\n";
					gridA.SetAt(CPoint (xsample,ysample), 
double(nodata));

					data = true;
				}


				maxout = (maxout + 1);


				if (maxout == 0)
				{
					data = true;
					cout << "Zone 1 did not contain enough valid data 
points!!! \n";

					i = n;
				}
			}
		}



		//Zone 2

		llxwin = int((431450.46-llx)/res);
		llywin = (nrows - int((4833354.37-lly)/res));
		urxwin = int((469027.7-llx)/res);
		urywin = (nrows - int((4883055.12-lly)/res));


		for (i=0;i<n;i++)
		{
			data = false;
			maxout = 200000 * ((urxwin-llxwin)*(urywin-llywin));

			while (!data)
			{
				xsample = getrandom(llxwin,urxwin);
				ysample = getrandom(urywin,llywin);


				if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
				{
					outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
					outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
					gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

					data = true;
				}

				maxout = (maxout + 1);

				if (maxout == 0)
				{
					data = true;
					cout << "Zone 2 did not contain enough valid data
points!!! \n";
					i = n;
				}
			}
		}
```

```
        //Zone 3

        llxwin = int((469027.70-llx)/res);
        llywin = (nrows - int((4849736.33-lly)/res));
        urxwin = int((499998.65-llx)/res);
        urywin = (nrows - int((4916303.1-lly)/res));


        for (i=0;i<n;i++)
        {
            data = false;
            maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

            while (!data)
            {
                xsample = getrandom(llxwin,urxwin);
                ysample = getrandom(urywin,llywin);

                if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
                {
                    outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
                    outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
                    gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

                    data = true;
                }

                maxout = (maxout + 1);

                if (maxout == 0)
                {
                    data = true;
                    cout << "Zone 3 did not contain enough valid data
points!!! \n";
                    i = n;
                }
            }
        }

        //Zone 4

        llxwin = int((499998.65-llx)/res);
```

```
		llywin = (nrows - int((4866320.69-lly)/res));
		urxwin = int((535269.05-llx)/res);
		urywin = (nrows - int((4940833.53-lly)/res));


		for (i=0;i<n;i++)
		{
			data = false;
			maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

			while (!data)
			{
				xsample = getrandom(llxwin,urxwin);
				ysample = getrandom(urywin,llywin);

				if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
				{
					outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
					outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
					gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

					data = true;
				}

				maxout = (maxout + 1);

				if (maxout == 0)
				{
					data = true;
					cout << "Zone 4 did not contain enough valid data
points!!! \n";
					i = n;
				}
			}
		}

		//Zone 5

		llxwin = int((535269.05-llx)/res);
		llywin = (nrows - int((4871969.51-lly)/res));
		urxwin = int((561878.3-llx)/res);
		urywin = (nrows - int((4940833.53-lly)/res));
		for (i=0;i<n;i++)
		{
			data = false;
			maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

			while (!data)
			{
				xsample = getrandom(llxwin,urxwin);
				ysample = getrandom(urywin,llywin);

				if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
				{
					outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
					outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
					gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

					data = true;
				}

				maxout = (maxout + 1);

				if (maxout == 0)
				{
					data = true;
					cout << "Zone 5 did not contain enough valid data
points!!! \n";
					i = n;
				}
			}
		}

		//Zone 6

		llxwin = int((561878.3-llx)/res);
		llywin = (nrows - int((4905491.63-lly)/res));
		urxwin = int((587744.29-llx)/res);
		urywin = (nrows - int((4940833.53-lly)/res));


		for (i=0;i<n;i++)
		{
			data = false;
```

```
            maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

            while (!data)
            {
                xsample = getrandom(llxwin,urxwin);
                ysample = getrandom(urywin,llywin);

                if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
                {
                    outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
                    outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
                    gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

                    data = true;
                }

                maxout = (maxout + 1);

                if (maxout == 0)
                {
                    data = true;
                    cout << "Zone 6 did not contain enough valid data
points!!! \n";
                    i = n;
                }
            }
        }

        //Zone 7

        llxwin = int((587744.29-llx)/res);
        llywin = (nrows - int((4905784.11-lly)/res));
        urxwin = int((617253.62-llx)/res);
        urywin = (nrows - int((4950690.77-lly)/res));


        for (i=0;i<n;i++)
        {
            data = false;
            maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

            while (!data)
            {
```

```
                        xsample = getrandom(llxwin,urxwin);
                        ysample = getrandom(urywin,llywin);

                        if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
                        {
                                outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
                                outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
                                gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

                                data = true;
                        }

                        maxout = (maxout + 1);

                        if (maxout == 0)
                        {
                                data = true;
                                cout << "Zone 7 did not contain enough valid data
points!!! \n";
                                i = n;
                        }
                }
        }

        //Zone 8

        llxwin = int((617253.62-llx)/res);
        llywin = (nrows - int((4917368.44-lly)/res));
        urxwin = int((662104.01-llx)/res);
        urywin = (nrows - int((4963889.58-lly)/res));


        for (i=0;i<n;i++)
        {
                data = false;
                maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

                while (!data)
                {
                        xsample = getrandom(llxwin,urxwin);
                        ysample = getrandom(urywin,llywin);

                        if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
```

```
					{
						outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
						outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
						gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

						data = true;
					}

					maxout = (maxout + 1);

					if (maxout == 0)
					{
						data = true;
						cout << "Zone 8 did not contain enough valid data
points!!! \n";
						i = n;
					}
				}
			}

			//Zone 9

			llxwin = int((637903.62-llx)/res);
			llywin = (nrows - int((4963889.58-lly)/res));
			urxwin = int((662104.01-llx)/res);
			urywin = (nrows - int((4985552.27-lly)/res));


			for (i=0;i<n;i++)
			{
				data = false;
				maxout = 1000 * ((urxwin-llxwin)*(urywin-llywin));

				while (!data)
				{
					xsample = getrandom(llxwin,urxwin);
					ysample = getrandom(urywin,llywin);

					if (gridA.GetAt(CPoint (xsample,ysample)) != nodata)
					{
						outFile << (xsample*res+llx) << "\t" << ((nrows -
ysample)*res+lly) << "\t";
```

```
                    outFile << gridA.GetAt(CPoint (xsample,ysample))
<< "\n";
                    gridA.SetAt(CPoint (xsample,ysample),
double(nodata));

                    data = true;
                }

                maxout = (maxout + 1);

                if (maxout == 0)
                {
                    data = true;
                    cout << "Zone 9 did not contain enough valid data
points!!! \n";
                    i = n;
                }
            }
        }

        outFile.close();

    }



    for (sampleloop=0;sampleloop<runs;sampleloop++)
    {
        filename = "";
        filename = int_to_string((sampleloop+1),filename);
        filename = "mean_result" + filename + ".asc";


        filename2 = "";
        filename2 = int_to_string((sampleloop+1),filename2);
        filename2 = "sample_set" + filename2 + ".txt";


        while (!observations.IsEmpty())
        {
            observations.RemoveTail();
        }

        for (i=0;i<nrows;i++)
        {
            for (j=0;j<ncols;j++)
```

```
		{
			tempgrid.SetAt (CPoint (j,i),nodata);
		}
	}


	inFile.open(filename2);
	if(!inFile)

	{
		cout << "Error opening file\n";
		return nRetCode;
	}

	while(inFile)
	{
		inFile >> sample.x >> sample.y >> sample.urchincount;
		if (inFile) observations.AddTail(sample);
	}

	inFile.close();


	POSITION pos = observations.GetHeadPosition();

	for (i=windowsize;i<(nrows-windowsize);i++)
	{
		for (j=windowsize;j<(ncols-windowsize);j++)
		{

			urchins = 0;
			count = 0;

			wincenterx = llx + j*res + 0.5*res;
			wincentery = lly + nrows*res - (i*res + 0.5*res);
			winllx = wincenterx - (0.5*res + windowsize*res);
			winlly = wincentery - (0.5*res + windowsize*res);
			winurx = wincenterx + (0.5*res + windowsize*res);
			winury = wincentery + (0.5*res + windowsize*res);

			pos = observations.GetHeadPosition();

			for (k=0;k<observations.GetCount();k++)
			{
				tempobs = observations.GetNext(pos);
```

```
                                        if ((tempobs.x >= winllx) && (tempobs.x < winurx)
&& (tempobs.y >= winlly) && (tempobs.y < winury))
                                        {
                                                d = sqrt(((wincenterx -
tempobs.x)*(wincenterx - tempobs.x)) + ((wincentery - tempobs.y)*(wincentery -
tempobs.y)));

                                                if (d <= tau)
                                                {
                                                        kf = (3/pi)*((1-((d/tau)*(d/tau)))*(1-
((d/tau)*(d/tau))));
                                                        urchins = urchins + (kf *
tempobs.urchincount);
                                                        count = count + kf;
                                                }
                                        }
                                }

                                if ((validgrid.GetAt(CPoint (j,i)) != nodata))
                                {
                                        if (count == 0)
                                                tempgrid.SetAt(CPoint (j,i),nodata);

                                        else
                                                tempgrid.SetAt(CPoint (j,i),(urchins/count));
                                }

                        }
                }


                outFile.open(filename);
                if(!outFile)
                {
                        cout << "Error opening file\n";
                        return nRetCode;
                }

                outFile << "NCOLS " << ncols << "\n";
                outFile << "NROWS " << nrows << "\n";
                outFile << "XLLCORNER " << llx << "\n";
                outFile << "YLLCORNER " << lly << "\n";
                outFile << "CELLSIZE " << res << "\n";
                outFile << "NODATA_VALUE " << nodata << "\n";
```

```
		for (i=0;i<nrows;i++)
			{
				for (j=0;j<ncols;j++)
				{
					outFile << tempgrid.GetAt(CPoint (j,i));
					outFile << " ";
				}

				outFile << "\n";
		}

		outFile.close();

	}


	cout << "\n";
	cout << "The output is stored in 40 files: \n" << "    20 with sample locations and
20 resulting averages";
	cout << "\n";
	cout << "enter 1 to finish\n";
	cin >> intdummy;

	return nRetCode;
}
```

<u>C++ computer code for generating plots of mean squared error (MSE) and mean MSE</u>

```
//Calculates MSE.  Arc ASCII gridA is the original density file, arc ASCII tempgrid is //the
simulated density file, and arc ASCII grid B is the depth and habitat constraints


#include "stdafx.h"
#include "biomass.h"
#include <fstream.h>
#include "math.h"
#include "Matrix.h"
#include "Location.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
```

```
#endif



CString int_to_string(int number, CString startstring)
{
	bool done = false;

	while (!done)
	{
		if ((number/10) < 1)
		{
			if (number == 0)
				startstring = "0" + startstring;
			if (number == 1)
				startstring = "1" + startstring;
			if (number == 2)
				startstring = "2" + startstring;
			if (number == 3)
				startstring = "3" + startstring;
			if (number == 4)
				startstring = "4" + startstring;
			if (number == 5)
				startstring = "5" + startstring;
			if (number == 6)
				startstring = "6" + startstring;
			if (number == 7)
				startstring = "7" + startstring;
			if (number == 8)
				startstring = "8" + startstring;
			if (number == 9)
				startstring = "9" + startstring;

			done = true;
		}

		if ((number/10) >= 1)
		{
			startstring = int_to_string((number-(int(number/10)*10)),
startstring);
			number = int(number/10);
		}
	}

	return startstring;
```

```
}


/////////////////////////////////////////////////////////////////////////////
// The one and only application object

CWinApp theApp;

//using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
	int nRetCode = 0;

	// initialize MFC and print and error on failure
	if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
	{
		// TODO: change error code to suit your needs
		cerr << _T("Fatal Error: MFC initialization failed") << endl;
		return nRetCode = 1;
	}
//////////////////////////////////////////////////////////////////////////////
//////// MY CODE


	ifstream inFile;  // Input data file.
	ofstream outFile; // Output data file.
	CMatrix gridA, tempgrid, result;
	char chardummy;
	int ncols, nrows;
	int intdummy = 0, i, j, nodata=-9999;
	double res, doubledummy, llx, lly;
	CString filename;
	int sampleloop;
	double a, b, c;


	gridA.Empty();
	tempgrid.Empty();

	int runs = 2;

	cout << "Enter 1 to load gridA.asc (will be used for llx, lly, and resolution): \n";
	cin >> intdummy;
	cout << " \n";
	inFile.open("gridA.asc");
	if(!inFile)

	{
		cout << "Error opening file\n";
		return nRetCode;
	}

	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy;
	inFile >> ncols;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy;
	inFile >> nrows;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> llx;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> lly;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> res;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy;
	inFile >> nodata;

	gridA.SetMatrixSize(CSize (ncols, nrows));


	for (i=0;i<nrows;i++)
		{
			for (j=0;j<ncols;j++)
			{
				inFile >> doubledummy;
				gridA.SetAt(CPoint (j,i), doubledummy);
			}
		}

	inFile.close();


	tempgrid.SetMatrixSize(CSize (ncols,nrows));
	result.SetMatrixSize(CSize (ncols,nrows));
	for (i=0;i<nrows;i++)
			{
				for (j=0;j<ncols;j++)
				{
					result.SetAt(CPoint (j,i), 0.0);
				}
			}



	for (sampleloop=0;sampleloop<runs;sampleloop++)
	{

		filename = "";
		filename = int_to_string((sampleloop+1),filename);
		filename = "mean_result" + filename + ".asc";


		inFile.open(filename);
		if(!inFile)

		{
			cout << "Error opening file\n";
			return nRetCode;
		}

		inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy;
		inFile >> intdummy;
		inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy;
		inFile >> intdummy;
		inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
		inFile >> doubledummy;
		inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
		inFile >> doubledummy;
		inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy;
		inFile >> doubledummy;
		inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy;
		inFile >> intdummy;
		for (i=0;i<nrows;i++)
			{
				for (j=0;j<ncols;j++)
				{
					inFile >> doubledummy;
					tempgrid.SetAt(CPoint (j,i), doubledummy);
				}
			}

		inFile.close();



		for (i=0;i<nrows;i++)
			{
				for (j=0;j<ncols;j++)
				{
					if (tempgrid.GetAt(CPoint (j,i)) != nodata)
					{
						a = tempgrid.GetAt(CPoint (j,i));
						b = gridA.GetAt(CPoint (j,i));
						c = (b-a)*(b-a);
						c = c + result.GetAt(CPoint (j,i));
						result.SetAt(CPoint (j,i), c);
					}
				}
			}

	}

	for (i=0;i<nrows;i++)
			{
				for (j=0;j<ncols;j++)
				{
					if (result.GetAt(CPoint (j,i)) != nodata)
					{
						a = (result.GetAt(CPoint (j,i))/runs);
						result.SetAt(CPoint (j,i), a);
					}
				}
			}


	outFile.open("output.asc");
	if(!outFile)
	{
		cout << "Error opening file\n";
		return nRetCode;
	}

	outFile << "NCOLS " << ncols << "\n";
	outFile << "NROWS " << nrows << "\n";
	outFile << "XLLCORNER " << llx << "\n";
	outFile << "YLLCORNER " << lly << "\n";
	outFile << "CELLSIZE " << res << "\n";
	outFile << "NODATA_VALUE " << nodata << "\n";


	for (i=0;i<nrows;i++)
		{
			for (j=0;j<ncols;j++)
			{
				outFile << result.GetAt(CPoint (j,i));
				outFile << " ";
			}

			outFile << "\n";
		}

	outFile.close();


	cout << "\n";
	cout << "The output is stored in output.asc";
	cout << "\n";
	cout << "enter 1 to finish\n";
	cin >> intdummy;

	return nRetCode;
}


/*

	tempgrid.Empty();    // open and read grid


	inFile.open("somegrid.txt");
	if(!inFile)
	{
		cout << "Error opening file\n";
		return nRetCode;
	}

	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy;
	inFile >> intdummy;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy;
	inFile >> intdummy;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> doubledummy;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> doubledummy;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy;
	inFile >> doubledummy;
	inFile >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy >> chardummy >> chardummy >>
chardummy >> chardummy >> chardummy;
	inFile >> intdummy;


	for (i=0;i<nrows;i++)
		{
			for (j=0;j<ncols;j++)
			{
				inFile >> doubledummy;
				tempgrid.SetAt(CPoint (j,i), doubledummy);
			}
		}

	inFile.close();


*/
```