PBSmapping 2.63: User's Guide

Jon T. Schnute, Nicholas M. Boers, Rowan Haigh, and Alex Couture-Beil

Fisheries and Oceans Canada
Science Branch, Pacific Region
Pacific Biological Station
3190 Hammond Bay Road
Nanaimo, British Columbia
V9T 6N7

2012

User's Guide Revised from Canadian Technical Report of Fisheries and Aquatic Sciences 2549

Fisheries and Oceans Canada | Pêches et Océans Canada

Canada

Revisions to: Cat. No. Fs97-6/2549E ISSN 0706-6457

Last update: Nov 22, 2012

Correct citation for this publication:

Schnute, J.T., Boers, N.M., Haigh, R., and Couture-Beil, A. 2012. `PBSmapping` 2.63: user's guide revised from Canadian Technical Report of Fisheries and Aquatic Sciences 2549: vi + 114 p. Last updated Nov 22, 2012.

TABLE OF CONTENTS

Abstract iii
Résumé iii
Preface iv
1. Introduction 1
1.1. Software Installation 2
2. PBSmapping: Functions and Data 3
2.1. Data Structures for Maps 3
PolySet 3
PolyData 4
EventData 5
LocationSet 5
2.2. Map Projections 5
2.3. PBSmapping Functions and Algorithms 8
Import Functions 8
Graphics Functions 9
Computational Functions 10
Associating Points with Polygons 13
Set Theoretic Operations 14
2.4. Shoreline Data 15
2.5. Bathymetry Data 16
2.6. Examples and Applications 17
2.7. Strengths, Limitations, and Alternatives 21
3. Command-line Utilities 23
3.1. clipPolys.exe (Clip Polygons) 23
3.2. convUL.exe (Convert between UTM and LL) 23
3.3. findPolys.exe (Points-in-Polygons) 24
Acknowlegements 24
References 25
Appendix A. PBSdata package 27
Appendix B. Bathymetry Data 29
Appendix C. Generic Mapping Tools (GMT) 30
Appendix D. Source Code for Figures 34
Appendix E. PBSmapping Function Dependencies 40
Appendix F. PBSmapping Functions and Data 43

LIST OF TABLES

Table 1. Principal graphics functions in the PBSmapping package 9
Table 2. PolySets derived from GSHHS databases 16
Table A1. Data sets available in PBSdata 27
Table F1. Functions and data sets defined in PBSmapping 43

LIST OF FIGURES

Figure 1. Map of the world 6
Figure 2. Map of the northeastern Pacific Ocean (longitude-latitude) 7
Figure 3. Map of the northeastern Pacific Ocean (UTM easting-northing) 8
Figure 4. Illustration of the thinPolys function 13
Figure 5. Example of the joinPolys logic operations 14
Figure 6. Polylines created by contourLines and convCP 17
Figure 7. Tow tracks from a longspine thornyhead survey in 2001 18
Figure 8. Areas of islands in the southern Strait of Georgia 19
Figure 9. Pacific ocean perch survey data (1966-89) 20
Figure 10. Proof of Pythagoras' Theorem 21
Figure C1. PBSmapping compared with GMT – Vancouver Island 31
Figure C2. PBSmapping compared with GMT – tow tracks 33

ABSTRACT

Schnute, J.T., Boers, N.M., Haigh, R., and Couture-Beil, A. 2012. `PBSmapping` 2.63: user's guide revised from Canadian Technical Report of Fisheries and Aquatic Sciences 2549: vi + 114 p. Last updated Nov 22, 2012.

This report describes a second version of software designed to facilitate the compilation and analysis of fishery data, particularly data referenced by spatial coordinates. Our research stems from experiences with information on Canada's Pacific groundfish fisheries compiled at the Pacific Biological Station (PBS). Despite its origins in fishery data analysis, our software has broad applicability. The library `PBSmapping` extends the R-statistical language to include two-dimensional plotting features similar to those commonly available in a Geographic Information System (GIS). Embedded C code speeds algorithms from computational geometry, such as finding polygons that contain specified point events or converting between longitude-latitude and Universal Transverse Mercator (UTM) coordinates. We also present a number of convenient utilities for Microsoft Windows operating systems that support computational geometry outside the framework of R. Our results, which depend significantly on the work of students, illustrate the convergence of goals between academic training and applied research.

RÉSUMÉ

Schnute, J.T., Boers, N.M. Haigh, R., et Couture-Beil, A. 2012. `PBSmapping` 2.63: Guide de l'utilisateur révisé de Canadian Technical Report of Fisheries and Aquatic Sciences 2549: vi + 114 p. Dernier mis à jour Nov 22, 2012.

Le présent rapport décrit la seconde version du logiciel conçu pour faciliter la compilation et l'analyse de données halieutiques, en particulier les données référencées par des coordonnées spatiales. Nos travaux de recherche ont capitalisé sur des expériences menées à l'aide de données sur les pêches des poissons démersaux le long du littoral Pacifique du Canada, données compilées à la Station biologique du Pacifique (SBP). Bien que conçu initialement pour l'analyse de données halieutiques, notre logiciel peut s'appliquer à toute une variété de domaines. La bibliothèque `PBSmapping` *(Cartographie de la SBP)* étend le langage R pour inclure une capacité d'impression en deux dimensions semblable à celle habituellement disponible dans les systèmes d'information géographiques (SIG). Des modules en C permettent d'accélérer les algorithmes grâce à la géométrie numérique, en trouvant par exemple les polygones qui contiennent des événements ponctuels spécifiques ou en convertissant les longitudes et les latitudes en coordonnées de la projection transversale universelle (UTM). Nous présentons également un certain nombre d'applications intéressantes pour les systèmes d'exploitation Microsoft Windows, qui peuvent effectuer des opérations de géométrie numérique en dehors du cadre de travail R. Nos résultats, auxquels plusieurs étudiants ont grandement contribué, illustrent la convergence des objectifs de la formation académique et de la recherche appliquée.

PREFACE

During the last decade, I've had the pleasure of directing work by computer science students from various local universities. My research as a mathematician in fish stock assessment requires an extensive software toolkit, including statistical languages, compilers, and operating system utilities. It helps greatly to have bright, adaptive students who can learn new languages quickly, investigate software possibilities, answer technical questions, and design programs that assist scientific analysis. I'm particularly grateful for contributions from the following students:

- Robert Swan (University of Victoria), 1996;
- Mike Jensen (Malaspina University-College and Simon Fraser University), 1997 and 1999;
- Chris Grandin (Malaspina University-College), 2000 and 2001;
- Nick Henderson (Malaspina University-College), 2002;
- Nick Boers (Malaspina University-College), 2003-2006.
- Alex Couture-Beil (Malaspina University-College), 2005-2007

Starting in 1998, I began a formal connection with the Computing Science Department at Malaspina University-College (MUC). My discussions with faculty members, particularly Dr. Peter Walsh and Dr. Jim Uhl, highlighted the convergence of goals between academic training and scientific research. Projects designed for fish stock assessment give students an opportunity to further their computing science careers while producing useful software. Both MUC and the Pacific Biological Station (PBS), where I work, are located in Nanaimo, British Columbia, Canada. This happy juxtaposition makes it easy to engage students in the exchange of ideas between academia and applied research. For example, Jim Uhl participated directly in Nick Boers' PBS work term during the summer of 2003. Nick had completed a course in computer graphics taught by Jim in the fall of 2002. Algorithms in the textbook (Foley et al. 1996) proved invaluable for writing software to produce maps of the British Columbia coast with related fishery information.

Quantitative fishery science requires a strong connection between theory and practice. In his book on computing theory, Michael Sipser (1997, p. xii) tells students that:

> ". . . theory is good for you because studying it expands your mind. Computer technology changes quickly. Specific technical knowledge, though useful today, becomes outdated in just a few years. Consider instead the abilities to think, to express yourself clearly and precisely, to solve problems, and to know when you haven't solved a problem. These abilities have lasting value. Studying theory trains you in these areas."

While dealing with the issues addressed here, I found myself asking simple questions that have numerically interesting answers. How do you locate fishing events within management areas or other polygons? How should regional boundaries on maps be clipped to lie within a smaller rectangle? I soon realised that I had touched upon the emerging field of computational geometry, where people have devised clever and efficient algorithms for addressing such questions.

Remarkably effective software can now be obtained freely from the Internet. I'm particularly fond of R, a version of the powerful statistical language S (and later S-PLUS) devised by Becker et al. (1988). Venables and Ripley (1999, 2000) give excellent guidance for

using either language. Although written originally for Unix, R has also been implemented for Microsoft's Windows operating systems. The web site http://cran.r-project.org/ describes R as GNU S, "a freely available language and environment for statistical computing and graphics". The GNU project (http://www.gnu.org/), where the recursive acronym GNU means "GNU's Not Unix", offers a wealth of free software including compilers for C/C++, Fortran, and Pascal. Code can be written in these compiled languages to speed computations that would otherwise run more slowly in R. Nick Boers has used such linkages intelligently to bring fast computational geometry into our R-package `PBSmapping`.

To some extent, this report constitutes a second edition of an earlier report (Schnute et al. 2003) that describes a suite of software utilities developed at PBS. In particular, the package `PBSmapping` has undergone extensive renovations and improvements, and this document provides a definitive manual for using version 2. To accommodate the new material presented here, my co-authors and I have decided to remove sections of the earlier report that discuss other PBS software utilities, free software available on the Internet, and related technical information. Readers of this current report may also wish to acquire the earlier version for additional material not included here.

I want to mention two milestones achieved during the production of `PBSmapping`, Version 2. First, we have posted the current software as a contributed package on the Comprehensive R Archive Network (CRAN, http://cran.r-project.org/). Thanks to a remarkable collection of Perl scripts developed for the R project, source code in both C and R, along with suitable documentation files, can be tested and compiled automatically for distribution as both source and binary packages. Nick Boers ensured that our source materials met the necessary standards, and (after we made minor changes in the C code to avoid compiler warnings) the authors of the CRAN web site in Vienna, Austria accepted our contribution. Second, Nick applied to the Canadian Natural Sciences and Engineering Research Council (NSERC) for a grant to support graduate studies in computing science. His application cited his successful experience developing `PBSmapping`, Version 1, as documented in Schnute et al. (2003). To the delight of Nick's supporters at PBS and MUC, he won a substantial award, in fact the only NSERC grant given to a student from MUC this year. Congratulations, Nick, from your colleagues at PBS and professors at MUC. We'll follow your career at the University of Alberta in Edmonton with great interest.

Jon Schnute

This page has been left intentionally blank for printing purposes.

1. INTRODUCTION

This report describes software written to facilitate the compilation and analysis of fishery data, particularly data referenced by spatial coordinates. Our work developed from experiences constructing databases that capture information from Canada's Pacific groundfish fisheries. Fishing events take place across a broad range of coastal waters and result in the capture of many species. Initially, we focused on issues related to database design and development, as described in previous reports by Schnute et al. (1996), Haigh and Schnute (1999), Rutherford (1999), Schnute et al. (2001, Section 2 and Appendix A), and Sinclair and Olsen (2002). Analyses of these databases shifted our attention to the problem of portraying and understanding such complex information. Maps with statistical information proved especially useful, and we found ourselves facing questions commonly addressed by Geographic Information Systems (GIS).

Commercial GIS packages can be expensive, with an additional requirement for specialized training. Because analysts who deal with Pacific groundfish data often have experience using the statistical languages R (http://cran.r-project.org/, available for free) or S-PLUS (http://en.wikipedia.org/wiki/S-PLUS, available commercially), we began by writing bilingual functions for these languages to produce the maps required. Schnute et al. (2003) describe the package `PBSmapping`, Version 1, which evolved from these early experiences. After another year of development, we extensively revised the software, and Schnute et al. (2004) presented a user's manual for `PBSmapping`, Version 2. Subsequently, we have dropped the bilingual (R/S-PLUS) nature of `PBSmapping`, producing revisions solely for R, and now refer to the package as `PBSmapping` rather than 'PBS Mapping' used in earlier documents. Additionally, we maintain most of our PBS packages, including `PBSmapping`, at http://code.google.com/p/pbs-software/.

Section 2 covers the mapping software itself, which contains functions that perform numerous calculations on polygons. These include standard set theoretic operations (union, intersection, difference, exclusive-or), clipping, thinning, thickening, testing convexity, forming the convex hull, and calculating various statistics (such as mean, centroid, and area). We discuss public data that represent shorelines and ocean bathymetry, and the package includes sample data sets drawn from these sources. We also discuss the Universal Transverse Mercator (UTM) projection that gives a particularly accurate flat projection of the earth's surface. Our software can convert between longitude-latitude and UTM coordinates.

Section 3 documents a number of convenient command-line utilities, compiled separately from C code written for the `PBSmapping` package. These make it possible to perform some of the polygon functions outside the framework of R. Appendices provide additional information about various topics related to `PBSmapping`, including

A. a package (`PBSdata`) of supplementary information for `PBSmapping`;
B. an Internet source for global bathymetry data;
C. alternative Generic Mapping Tools (GMT);
D. source code for the figures in this report;
E. function dependencies in `PBSmapping`;

F. documentation for `PBSmapping` functions and data, including an indexed manual based on the `*.Rd` files.

We anticipate that our software will continue to change for the better, due to bug fixes and other improvements. This report documents version 2.63, which currently appears as a contributed package on the R archive (http://cran.r-project.org/). We will post subsequent versions as they become available. All software required to develop and use `PBSmapping` is freely available from the Internet.

1.1. Software Installation

We provide two mapping packages:

- `PBSmapping` – the mapping software discussed in Section 1;
- `PBSdata` – various additional data sets relevant to fisheries investigated at PBS (Appendix A).

Installation of `PBSmapping` can be achieved in two ways – (1) navigate to: http://cran.r-project.org/web/packages/PBSmapping/index.html, download the appropriate binary, and install from R using the menu <Packages><Install package(s) from local zip files...>, or (2) in R, use the menu <Packages><Install package(s)>, choose a CRAN mirror near you, highlight `PBSmapping`, and press OK. Note that the software is available in two forms:

- `PBSmapping_2.62.tar.gz` – source code for the R distribution, which can be used to build a binary package;
- `PBSmapping_2.62.zip` – binary package ready for installation into R;

The package `PBSdata` remains available to Fisheries and Oceans Canada personnel for installation from a local zip file (`PBSdata.zip`), which can be downloaded from the PBS Intranet website: http://svbcpbsgfiis/sql/. Look for a link entitled “PBS Data for the PBSmapping Package”.

To remove `PBSmapping` from R, open the `library\` directory and delete the associated subdirectory `PBSmapping\`. Before loading a new version of a package, we recommend the removal of any previous version. Eventually, the installation files may have names that reflect a version number later than the current version.

Additionally two other PBS packages are available from CRAN that facilitate fisheries analysis and research:

- `PBSmodelling` – http://cran.r-project.org/web/packages/PBSmodelling/index.html;
- `PBSddesolve` – http://cran.r-project.org/web/packages/PBSddesolve/index.html.

The `PBSmodelling` library includes a directory called `PBStools` that contains useful batch files for building R packages and generating an indexed manual based on the `*.Rd` files.

2. PBSmapping: FUNCTIONS AND DATA

Niklaus Wirth, the author of Pascal and Modula-2, summarises the essence of software design in the title of his book *Algorithms + Data Structures = Programs* (Wirth 1975). Our software package `PBSmapping` begins with data structures that embody two essential concepts. First, polygons define boundaries, such as shorelines and fishery management areas. Second, fishing events occur at specific locations defined by two geographical coordinates, such as longitude and latitude. The R language conveniently supports such structures through the concept of a *data frame*, essentially a database table in which rows and columns define records and fields, respectively. Objects like data frames in R can also have *attributes* that store additional properties, such as the projection used in defining a geographic coordinate system.

2.1. Data Structures for Maps

`PBSmapping` introduces four data structures, each stored as a data frame. Field names, attributes, and other properties of these objects implicitly dictate their type. An object may also identify its type explicitly in the `class` attribute. Each type requires a particular structure, as outlined below.

PolySet

In our software, a *PolySet* data frame defines a collection of polygonal contours (i.e., line segments joined at vertices), based on four or five numerical fields:

- `PID` – the primary identification number for a contour;
- `SID` – (optional) the secondary identification number for a contour;
- `POS` – the position number associated with a vertex;
- `X` – the horizontal coordinate at a vertex;
- `Y` – the vertical coordinate at a vertex.

The simplest PolySet lacks an `SID` column, and each `PID` corresponds to a different contour. By analogy with a child's "follow the dots" game, the `POS` field enumerates the vertices to be connected by straight lines. Coordinates (`X`, `Y`) specify the location of each vertex. Thus, in familiar mathematical notation, a contour consists of n points (x_i, y_i) with $i = 1, \ldots, n$, where i corresponds to the `POS` index. A PolySet has two potential interpretations. The first associates a line segment with each successive pair of points from 1 to n, giving a *polyline* (in GIS terminology) composed of the sequential segments. The second includes a final line segment joining points n and 1, thus giving a *polygon*.

The secondary ID field allows us to define regions as composites of polygons. From this point of view, each primary ID identifies a collection of polygons distinguished by secondary IDs. For example, a single management area (`PID`) might consist of two fishing areas, each associated with a different `SID`. A secondary polygon can also correspond to an inner boundary, like the hole in a doughnut. We adopt the convention that `POS` goes from 1 to n along an outer boundary, but from n to 1 along an inner boundary, regardless of rotational direction. This

contrasts with other GIS software, such as ArcView (ESRI 1996), in which outer and inner boundaries correspond to clockwise and counter-clockwise directions, respectively.

The SID field in a PolySet with secondary IDs must have integer values that appear in ascending order for a given PID. Furthermore, inner boundaries must follow the outer boundary that encloses them. The POS field for each contour (PID, SID) must similarly appear as integers in strictly increasing or decreasing order, for outer and inner boundaries respectively. If the POS field erroneously contains floating-point numbers, fixPOS can renumber them as sequential integers, thus simplifying the insertion of a new point, such as point 3.5 between points 3 and 4.

A PolySet can have a projection attribute, which may be missing, that specifies a map projection. In the current version of PBSmapping, projection can have character values "LL" or "UTM", referring to "Longitude-Latitude" and "Universal Transverse Mercator". We explain these projections more completely below. If projection is numeric, it specifies the aspect ratio *r*, the number of *x* units per *y* unit. Thus, *r* units of *x* on the graph occupy the same distance as one unit of *y*. Another optional attribute zone specifies the UTM zone (if projection="UTM") or the preferred zone for conversion from Longitude-Latitude (if projection="LL").

A data frame's class attribute by default contains the string "data.frame". Inserting the string "PolySet" as the class vector's first element alters the behaviour of some functions. For example, the summary function will print details specific to a PolySet. Also, when PBSprint is TRUE, the print function will display a PolySet's summary rather than the contents of the data frame.

PolyData

We define *PolyData* as a data frame with a first column named PID and (optionally) a second column named SID. Unlike a PolySet, where each contour has many records corresponding to the vertices, a PolyData object must have only one record for each PID or each (PID, SID) combination. Conceptually, this object associates data with contours, where the data correspond to additional fields in the data frame. The R language conveniently allows data frames to contain fields of various atomic modes ("logical", "numeric", "complex", "character", and "null"). For example, PolyData with the fields (PID, PName) might assign character names to a set of primary polygons. Additionally, if fields X and Y exist (perhaps representing locations for placing labels), consider adding attributes zone and projection. Inserting the string "PolyData" as the class attribute's first element alters the behaviour of some functions, including print (if PBSprint is TRUE) and summary.

Our software particularly uses PolyData to set various plotting characteristics. Consistent with graphical parameters used by the R functions lines and polygon, column names can specify graphical properties:

- lty – line type in drawing the border and/or shading lines;
- col – line or fill colour;
- border – border colour;

- `density` – density of shading lines;
- `angle` – angle of shading lines.

When drawing polylines (as opposed to closed polygons), only `lty` and `col` have meaning.

EventData

We define *EventData* as a data frame with at least three fields named (`EID`, `X`, `Y`). Conceptually, an EventData object describes events (`EID`) that take place at specific points (`X`, `Y`) in two-dimensional space. Additional fields specify measurements associated with these events. For example, in a fishery context EventData could describe fishing events associated with trawl tows, based on the fields:

- `EID` – fishing event (tow) identification number;
- `X, Y` – fishing location;
- `Duration` – length of time for the tow;
- `Depth` – average depth of the tow;
- `Catch` – biomass captured.

Like PolyData, EventData can have attributes `projection` and `zone`, which may be absent. Inserting the string `"EventData"` as the `class` attribute's first element alters the behaviour of some functions, including `print` (if `PBSprint` is `TRUE`) and `summary`.

LocationSet

A PolySet can define regional boundaries for drawing a map, and EventData can give event points on the map. Which events occur in which regions? Our function `findPolys`, discussed in Section 2.3 below, solves this problem. The output lies in a *LocationSet*, a data frame with three or four columns (`EID`, `PID`, `SID`, `Bdry`), where `SID` may be missing. One row in a LocationSet means that the event `EID` occurs in the polygon (`PID`, `SID`). The boundary (`Bdry`) field specifies whether (`Bdry=T`) or not (`Bdry=F`) the event lies on the polygon boundary. If `SID` refers to an inner polygon boundary, then `EID` occurs in (`PID`, `SID`) only if `Bdry=T`. An event may occur in multiple polygons. Thus, the same `EID` can occur in multiple records. If an `EID` does not fall in any (`PID`, `SID`), or if it falls within a hole, it does not occur in the output LocationSet. Inserting the string `"LocationSet"` as the first element of a LocationSet's `class` attribute alters the behaviour of some functions, including `print` (if `PBSprint` is `TRUE`) and `summary`.

2.2. Map Projections

The simplest projection associates each point on the earth's surface with a longitude x ($-360° \le x \le 360°$) and latitude y ($-90° \le y \le 90°$), where $x = 0°$ on the Greenwich prime meridian. The chosen range of x depends on the region of interest, where negative longitudes refer to displacements west of the prime meridian. When plotted on a rectangular grid with equal distances for each degree of longitude and latitude, this projection exaggerates the size of objects

near the earth's poles, as illustrated in Figure 1. For points near the latitude y, a more realistic map uses the aspect ratio

$$r = \frac{1}{\cos y}, \tag{2.1}$$

where r degrees of longitude x should occupy the same distance as 1 degree of latitude y.

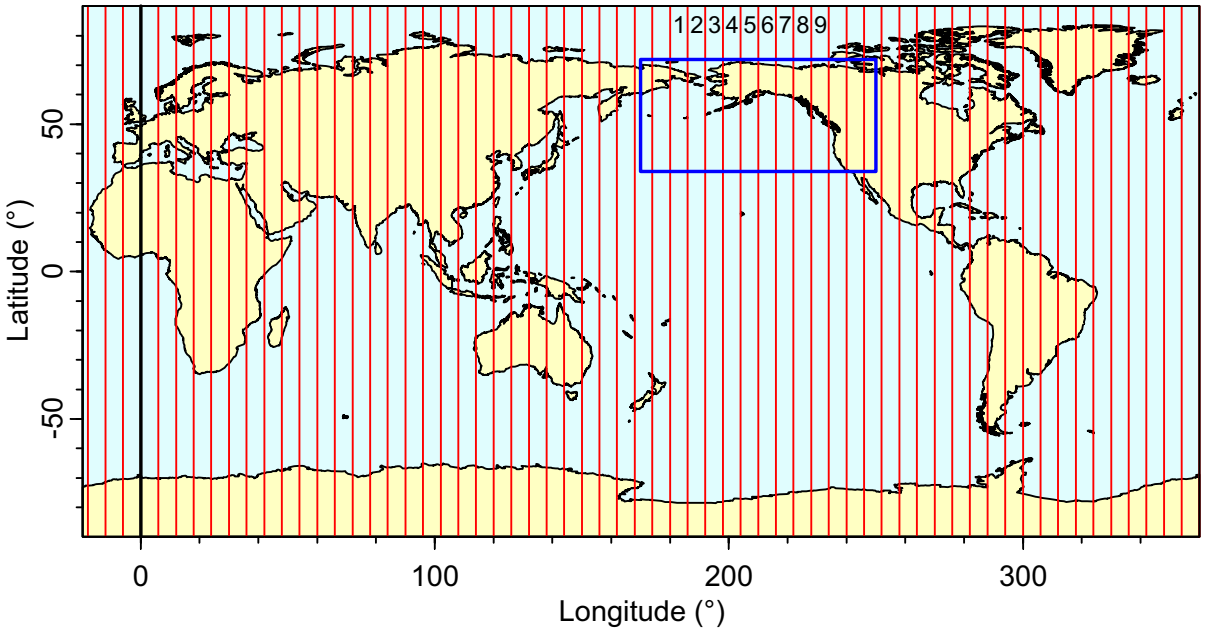

Figure 1. Map of the world projected in longitude-latitude coordinates. This image, based on our PolySet `worldLL`, uses the longitude range $-20° \le x \le 360°$ to produce a convenient cut in the eastern Atlantic Ocean. Red vertical lines show boundaries for the 60 Universal Transverse Mercator (UTM) zones, with explicit labels for zones 1 to 9. A black line indicates the prime meridian $(x = 0°)$. Our PolySet `nepacLL` lies within the clipping boundary shown as a blue rectangle.

The Universal Transverse Mercator (UTM) projection gives a more realistic portrayal of the earth's surface within 60 standardized longitude zones. Each zone spans $6°$, and zone i includes points with longitude x in the range

$$(-186 + 6i)° < x \le (-180 + 6i)° \qquad \text{[UTM zone } i\text{]} \tag{2.2}$$

The mid-longitude in (2.2)

$$x_i = (-183 + 6i)° \qquad \text{[Central meridian, zone } i\text{]} \tag{2.3}$$

defines the *central meridian* of zone i. In particular, zone 9 has central meridian $-129°$ and covers the range

$$-132° < x \le -126°. \qquad \text{[UTM zone 9]} \tag{2.3}$$

Canada's Pacific coast lies in zones 8 to 10 (Figure 2), and the projection to zone 9 gives a reasonably accurate map for fisheries in this region.

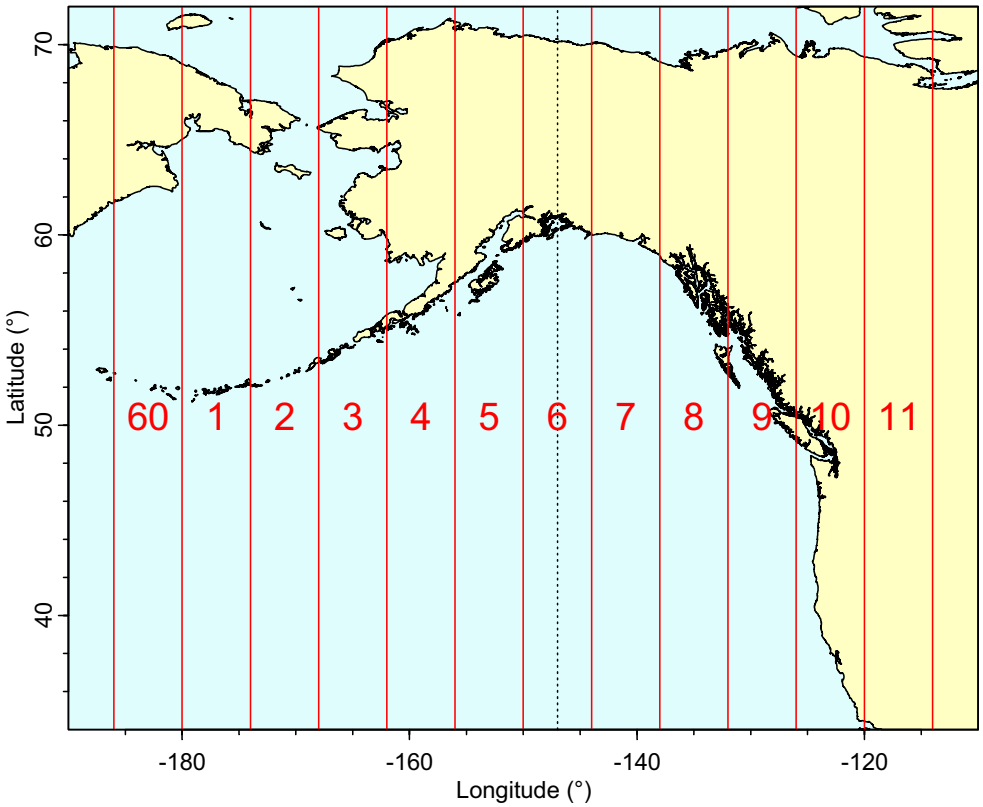

Figure 2. Shoreline data in longitude-latitude coordinates for the northeastern Pacific Ocean, as captured in our PolySet `nepacLL`. Vertical red lines display UTM boundaries for zones 60, 1, 2, …, 11. A vertical dotted line indicates the central meridian of zone 6, near the centre of this figure.

Visually, UTM zones look like sections of orange peel cut from top to bottom. Each relatively narrow section can be flattened without too much distortion to give coordinates (X, Y) measured as actual distances, as illustrated by zone 6 in Figure 3. Complex formulas, compiled in detail by the UK Ordnance Survey (Anonymous 1998, Ordnance Survey 2010), allow conversion between two projections: the UTM *easting-northing* coordinates (X, Y) and the usual longitude-latitude coordinates (x, y). These take account of the earth's ellipsoidal shape, with a wider diameter at the equator than the poles. The UTM projection scales distances exactly along two great circles: the equator and the central meridian, which act as X and Y axes, respectively. Along the equator, $Y = 0$ km by definition; elsewhere, Y indicates the distance north (positive Y) or south (negative Y) of the equator. The central meridian is assigned a standard easting $X = 500$ km, rather than the usual $X = 0$ km. This ensures that $X > 0$ km throughout the zone. In effect, the difference $X - 500$ km represents the distance east of the central meridian, where a negative distance corresponds to a westward displacement. These interpretations are exact along the equator and central meridian, but approximate elsewhere.

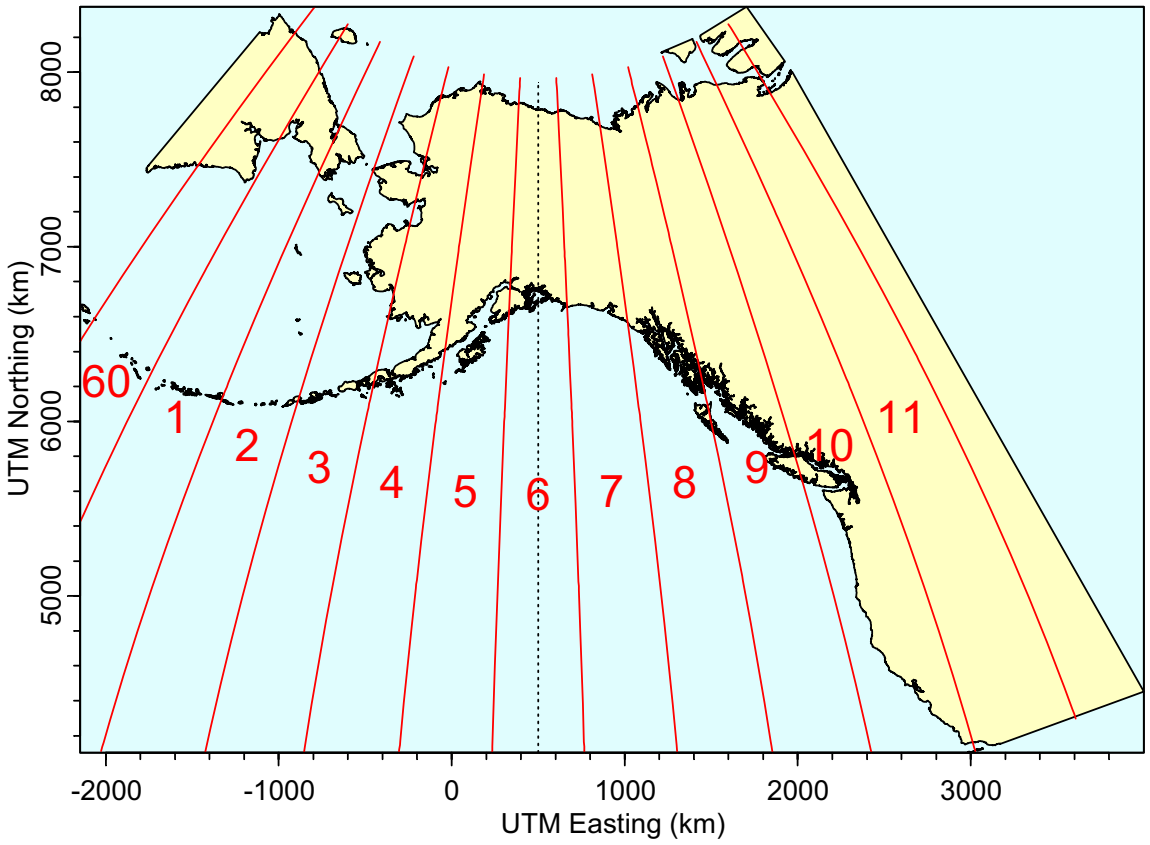

Figure 3. Shoreline data for the northeastern Pacific Ocean, projected in UTM coordinates (zone 6) from our PolySet `nepacLL`. Vertical red lines show UTM zone boundaries. The central axis of zone 6 (vertical dotted line at $x = 500$ km) corresponds to the central meridian shown in Figure 2.

2.3. `PBSmapping` Functions and Algorithms

Our software produces maps from the data structures defined in Section 2.1. Following typical design concepts in R, it uses functions to generate plots, implement algorithms, and perform other tasks. Where possible, function arguments often have explicit default values. `PBSmapping` includes many functions not mentioned in this section. We encourage readers to examine Appendix F, which gives detailed technical descriptions of all our software's functions and other components.

Import Functions

The following functions provide some support for importing GIS data from other users and other mapping platforms:

- `importEvents` import a text file and convert into EventData.
- `importLocs` import a text file and convert into a LocationSet.
- `importPolys` import a text file and convert into a PolySet with optional PolyData attribute.

- `importGSHHS` import data from a GSHHS database and convert data into a PolySet with a PolyData attribute. GSHHS: A Global Self-consistent, Hierarchical, High-resolution Shoreline Database, http://www.soest.hawaii.edu/wessel/gshhs/gshhs.html
  See Section 2.4 below for more details.
- `importShapefile` imports an ESRI shapefile (`.shp`) into either a PolySet or EventData. The function relies on C-code provided by Roger Bivand's package `maptools`.

Graphics Functions

In the R language, high-level commands (like `plot`) create new graphs; lower-level commands (like `points` and `lines`) add features to an existing graph. Similarly, we provide functions (`plotLines`, `plotMap`, `plotPoints`, `plotPolys`) that create graphs and others (`addLabels`, `addLines`, `addPoints`, `addPolys`, `addStipples`) that add graphical features.

Some of these plotting functions draw objects defined by a PolySet, while others expect EventData, a LocationSet, or PolyData. Both `plotLines` and `addLines` treat their input PolySet as polylines, with no connection between the last and first vertices. By contrast, `plotMap`, `plotPolys`, and `addPolys` regard their input as polygons, where a final line segment connects the last vertex to the first. The functions `plotMap` and `plotPolys` behave similarly, except that `plotMap`'s default behaviour guarantees the correct aspect ratio, as defined by either the PolySet's `projection` attribute or the function's `projection` argument. If both are specified, the attribute supersedes the argument. When this attribute is missing, `plotMap` uses a 1:1 projection. Table 1 summarises the default behaviour of our principal graphics commands. A user concerned with drawing maps, where the correct aspect ratio plays a key role, would likely initiate a graph with the `plotMap` function. However, `plotPolys`, `plotLines`, and `plotPoints` can also set the correct aspect ratio when passed a suitable `projection` argument.

Table 1. Behaviour of the principal graphics functions in the `PBSmapping` software package.

Function	Creates a Graph	Plots as Polygons	Sets Aspect Ratio by Default
`addLabels`	No	-	-
`addLines`	No	No	-
`addPoints`	No	-	-
`addPolys`	No	Yes	-
`addStipples`	No	-	-
`plotLines`	Yes	No	No
`plotMap`	Yes	Yes	Yes
`plotPoints`	Yes	-	No
`plotPolys`	Yes	Yes	No

Our high-level graphics functions accept a common set of arguments, consistent with existing `par` parameters where possible. These include

- `xlim` and `ylim` to specify horizontal and vertical coordinate ranges;

- `projection` to specify the projection used in drawing the map or graph;
- `plt` to define the plot region relative to the figure region;
- `polyProps` to support plotting properties for individual contours (Section 2.1);
- `lty`, `cex`, `col`, `border`, `density`, `pch`, and `angle` to adjust properties of labels, lines, points, and polygons where applicable;
- `axes` to disable plotting axes;
- `tck` to control (major) tick mark lengths;
- `tckMinor`, a counterpart of `tck`, to set a different length for minor tick marks;
- `tckLab`, with Boolean values, to determine whether to include numeric tick labels.

We introduce `tckMinor` and `tckLab` to give finer control over the appearance of tick marks. Each of `tck`, `tckLab`, and `tckMinor` can have length one or two. A single value pertains to both axes, and two values specify distinct parameters for the horizontal and vertical axes, respectively.

Our low-level graphics functions (e.g., `addLines`) use many of the same arguments as their high-level counterparts (e.g., `plotLines`). However, they do not accept parameters that affect the overall plot, such as `xlim`, `ylim`, `projection`, `plt`, `axes`, or any of the `tck` arguments.

The `par` parameter `plt` plays a special role in `PBSmapping`, because we use it to set the aspect ratio required for a particular `projection`. Recall that in R the plot region lies inside the figure region, which similarly lies inside the overall device region. The parameter `plt` specifies the plot region boundaries as fractions (left, right, bottom, top) of the current figure region. Our high-level plotting functions use the initial default value

```
plt=c(0.11,0.98,0.12,0.88),
```

but then alter `plt` by shrinking the width or height to achieve the required aspect ratio. In the function call, the argument `plt` can set a different default value, but again this may be changed by the graphics function to set the aspect ratio. In effect, the argument `plt` sets minimum margins for the plot within the figure region, but the aspect ratio may force the plot to shrink in width or height, giving wider margins in one direction.

Standard high-level commands in R (like `plot`) do not allow layout parameters (like `plt`) to be passed as arguments. Instead, users normally use `par` to set these parameters before invoking a graphics command. However, unlike normal graphics commands, those in `PBSmapping` actually alter the margins, so we adopt a different approach in which `plt` is reset with each high-level command. Advanced users wishing to set the plot region using the `par` parameters `mai` or `mar` can disable the default initial size with the argument `plt=NULL`.

Computational Functions

`PBSmapping` contains many functions that perform computations on PolySets and other data structures. Appendix F lists them all, but we give further details for some of them here,

including formulas or algorithms for implementation and references for further reading. In alphabetic order, this list below highlights key features of selected functions in the package.

- `calcArea` computes polygon areas by the formula (Rokne 1996)

$$A = \frac{1}{2}\sum_{i=1}^{n-1}\left(x_i y_{i+1} - x_{i+1} y_i\right),$$

for the area A of a polygon with vertices (x_i, y_i), $i = 1,\ldots,n$, where vertices 1 and n correspond to the same point: $(x_1, y_1) = (x_n, y_n)$. This formula assumes identical units for x and y (an aspect ratio 1), as in UTM coordinates. The function automatically converts longitude-latitude coordinates to UTM before calculating the area.

- `calcCentroid` computes polygon centroid coordinates (x, y) by the formulas (Bourke 1988)

$$x = \frac{1}{6A}\sum_{i=1}^{n-1}\left(x_i + x_{i+1}\right)\left(x_i y_{i+1} - x_{i+1} y_i\right)$$
$$y = \frac{1}{6A}\sum_{i=1}^{n-1}\left(y_i + y_{i+1}\right)\left(x_i y_{i+1} - x_{i+1} y_i\right)$$

for a polygon with vertices (x_i, y_i), $i = 1,\ldots,n$, where vertices 1 and n correspond to the same point: $(x_1, y_1) = (x_n, y_n)$ and A is computed by the formula shown above in the definition of `calcArea`. These formulas scale automatically to the units of x and y and consequently do not depend on the projection attribute.

- `calcConvexHull` calculates the convex hull for a given set of points using the function `chull()` in R's package `grDevices`.
- `calcLength` calculates polyline lengths using Pythagoras' Theorem when the projection is UTM or 1. Thus, the distance d between points (x, y) and (x', y') is

$$d = \sqrt{(x'-x)^2 + (y'-y)^2}.$$

The function also supports longitude-latitude coordinates (x, y) by calculating great circle distances between polygon vertices. In this case, the distance d between two points is (Chamberlain 2001)

$$d = 2R \arcsin\left[\sqrt{\sin^2\left(\frac{y'-y}{2}\right) + (\cos y)(\cos y')\sin^2\left(\frac{x'-x}{2}\right)}\right],$$

where $R = 6371.3$ km denotes the earth's mean radius (Wikipedia 2004).

- `calcMidRange` calculates midpoints of the `X` and `Y` ranges for each given polygon.
- `calcSummary` calculates summary statistics for a PolySet, given a user-defined function.
- `calcVoronoi` calculates the Voronoi (Dirichlet) tesselation for a set of points (using the `deldir` function from the package `deldir`) and creates a PolySet. See Figure 8 of the `PBSmodelling` user's guide (Schnute et al. 2006) for an example called `CalcVor`.

- `clipLines` (and `clipPolys`) clips polylines (and polygons) within a specified rectangle, possibly smaller than the bounding rectangle, using the Sutherland-Hodgman clipping algorithm (Foley et al. 1996, p. 124-127).
- `closePolys` adds corners from the bounding rectangle, if needed, to close polylines into polygons.
- `combinePolys` combines several polygons into a single polygon by modifying the PID and SID indices.
- `convCP` converts results from `contourlines` into a PolySet.
- `convDP` converts EventData/PolyData into a PolySet.
- `convLP` converts two polylines into a polygon.
- `convUL` converts between UTM and longitude-latitude coordinates using the extensive formulas presented in Ordnance Survey (2010).
- `dividePolys` divides a single polygon (with several outer-contour components) into several polygons, a polygon for each outer contour, by modifying the PID and SID indices.
- `findCells` finds the cells in a grid PolySet that contain events specified in EventData, using the "crossings test" algorithm described later in this section.
- `findPolys` finds the polygons in a PolySet that contain events specified in EventData, using the "crossings test" algorithm described later in this section.
- `isConvex` determines which polygons in a PolySet are convex, using an algorithm described below.
- `isIntersecting` finds polygons that self-intersect by comparing each edge pairwise with every other edge.
- `joinPolys` performs set theoretic operations (union, intersection, difference, and exclusive-or) on polygons using the General Polygon Clipper (GPC) library © Copyright 1997-2012, Advanced Interfaces Group, University of Manchester, developed by Murta (2004). See Figure 13 of the `PBSmodelling` user's guide (Schnute et al. 2006) for an example called `FishTows`.
- `thickenPolys` adds vertices to polygons using an algorithm described below.
- `thinPolys` thins the number of polygon vertices, based on the Douglas-Peuker line simplification algorithm (Douglas and Peucker 1973), as illustrated in Figure 4.

Our function `isConvex` first calls `isIntersecting` to determine whether or not a polygon self-intersects. If it does, it cannot be convex and the result is `FALSE`. Otherwise, the function proceeds. Three sequential points in a non-self-intersecting polygon describe a left turn, a straight line, or a right turn. The function locates the first non-straight turn (left or right) in a polygon and checks that all subsequent turns are either the same or straight. If so, the polygon is convex; otherwise it is not.

Like `calcLength`, `thickenPolys` also supports the longitude-latitude projection. In this case, `tol` is measured in kilometres and distances are computed along great circles (Chamberlain 2001).

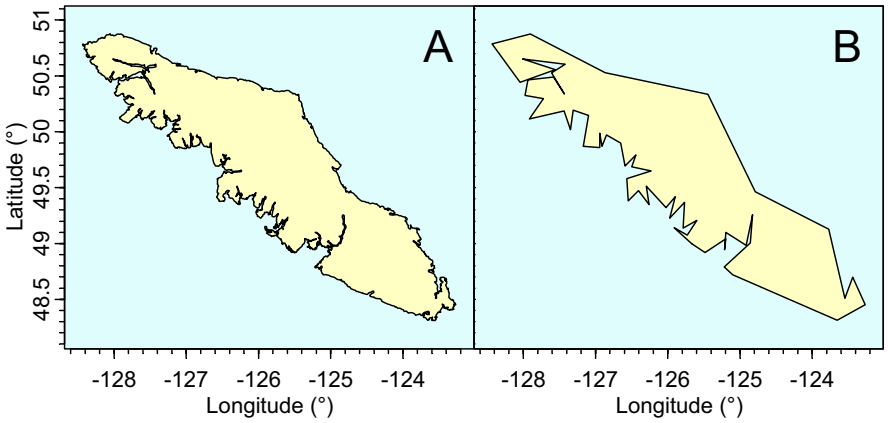

Figure 4. (A) Vancouver Island clipped from the PolySet `nepacLL` and (B) the result of applying `thinPolys` to this polygon with a tolerance of ten kilometres.

When the projection is UTM or 1, our function `thickenPolys` accepts a tolerance specified in `X` or `Y` units (kilometres in the UTM case). It operates in two distinct modes. When `keepOrig=TRUE`, it retains all original vertices and adds vertices, as required, along each edge. Thus, if the distance between two sequential original vertices exceeds the specified tolerance `tol`, it adds enough vertices spaced evenly between them so that sequential vertices lie at most the distance `tol` apart. When `keepOrig=FALSE`, the algorithm guarantees only that the first vertex of each polygon appears in the result. Starting at that vertex, the algorithm walks through the polygon while summing distances between vertices. When the cumulative distance exceeds `tol`, it adds a vertex on the line segment under inspection. It then resets the distance sum and continues walking the polygon from this new vertex.

Associating Points with Polygons

As discussed in the definition of LocationSet (Section 2.1), our function `findPolys` solves the "points-in-polygons" problem. Given a set of points (EventData) and a collection of polygons (a PolySet), which points lie in which polygons? Several algorithms solve this problem, including:

- **The crossings test.** Draw a ray from the trial point in a fixed direction (e.g., upward). If the ray crosses an even number of polygon edges, the point must be outside. For an inside point, the number of crossings must be odd.
- **The angle summation (or winding number) test.** Sum the angles swept by a ray from the trial point to sequential vertices of the polygon. For a point outside the polygon, the angles sum to 0 because the ray sweeps back and forth, returning to the starting point. For an inside point, the ray traces a full circle, and the angles do not sum to zero.

We use the crossings test because it performs faster than angle summation (Hains 1994, p. 26-27). The latter requires large numbers of trigonometric function calls.

After finding the polygons that contain various events, an analyst often wants to compute statistics associated with the events that occur inside each polygon. For example, in a fishery context, what is the total catch from all fishing events within each management region? Our

function `combineEvents` supports such calculations. The function `makeProps` can then relate polygon properties, such as colour used for plotting, to these computed statistical values.

Set Theoretic Operations

We include the function `joinPolys` to apply set theoretic operations (difference, intersection, union, and exclusive-or) to one or two PolySets. Our `joinPolys` function interfaces with the General Polygon Clipper (GPC) library © Copyright 1997-2012, Advanced Interfaces Group, University of Manchester, developed by Alan Murta (2004). We adopt some of his terminology in the discussion here. He defines a *generic polygon* (or *polygon set*) as zero or more disjoint polygonal contours that define boundaries of the polygon region. Some contours can represent inner boundaries that define holes in the region. Each contour can be convex, concave, or self-intersecting.

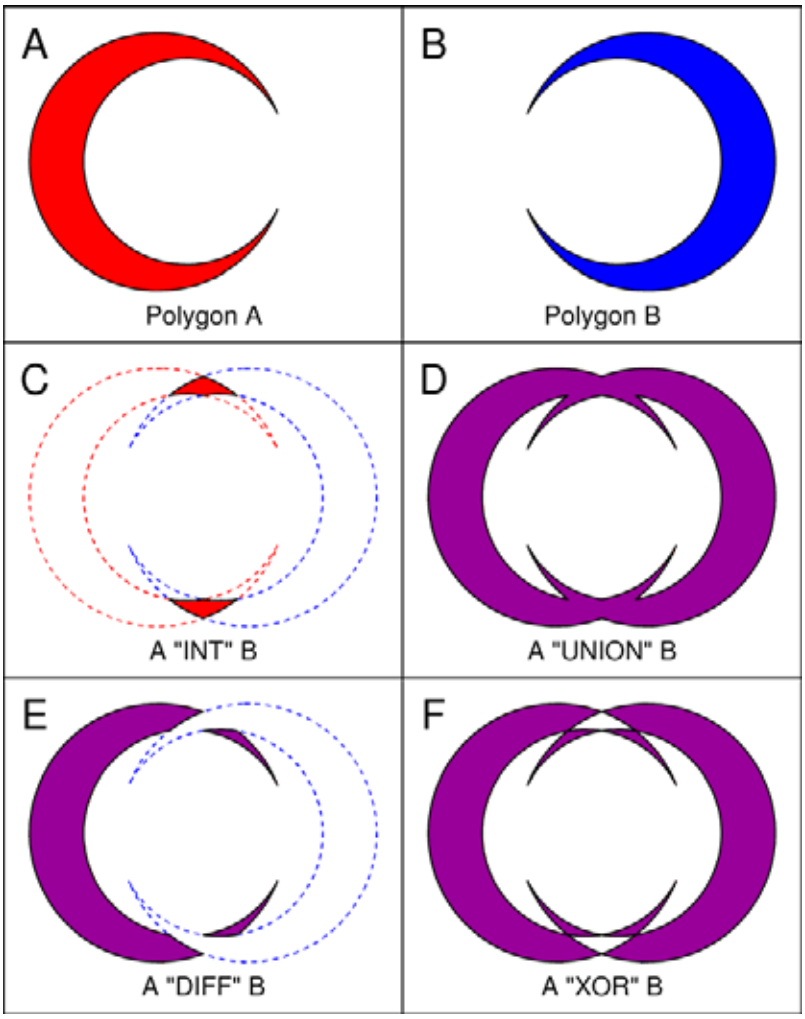

Figure 5. Example of the `joinPolys` logic operations. Panels A and B display the first and second PolySets, respectively. Panels C to F illustrate the intersection, union, difference, and exclusive-or operations, respectively.

In our PolySet, the polygons associated with each unique `PID` correspond to a generic polygon with some restrictions. Some of our functions do not support self-intersecting polygons. Furthermore, the `SID` contours cannot be arranged in arbitrary order because we require that hole contours follow the outer contours in which they lie.

The function `joinPolys` can also accept two PolySet arguments *P* and *Q*. In this case, the function returns a PolySet with all possible pairwise applications of *op* between generic polygons in *P* and *Q*. For example, if *P* contains (*A*, *B*, *C*) and *Q* contains (*D*, *E*), then `joinPolys` returns a PolySet with six PIDs corresponding to the six generic polygons *A op D*, *B op D*, *C op D*, *A op E*, *B op E*, and *C op E*. More generally, if *P* and *Q* include *m* and *n* generic polygons, respectively, then the function returns a PolySet with $m \times n$ generic polygons. If $m = 1$ or $n = 1$, the output preserves `PID`s from the PolySet with more than one generic polygon. Figure 5 illustrates the four supported set theoretic operations applied to crescent-shaped polygons *A* and *B*.

Applied to one PolySet *P*, our function `joinPolys` applies the set theoretic operation *op* sequentially to the generic polygons in *P*. For example, suppose that *P* contains three generic polygons (*A*, *B*, *C*). Then the function returns a PolySet containing the generic polygon ((*A op B*) *op C*), represented as one `PID` with possibly many `SID`s.

2.4. Shoreline Data

To portray fishery data along Canada's Pacific coast, we need a PolySet that defines the relevant shoreline. Originally, we began with a polyline of the British Columbia coast, digitized manually from a marine map. To convert this object to a meaningful closed polygon, we devised the functions `fixBound` and `closePolys`. Satellite imagery and other sources, however, make our initial coastline obsolete. For example, Wessel and Smith (1996) have used information from the public domain to assemble a Global Self-consistent, Hierarchical, High-resolution Shoreline (GSHHS, http://www.soest.hawaii.edu/wessel/gshhs/gshhs.html) database for the entire planet. They make this available via the Internet as binary files in five different resolutions: full (`gshhs_f.b`), high (`gshhs_h.b`), intermediate (`gshhs_i.b`), low (`gshhs_l.b`), and crude (`gshhs_c.b`). They also supply software as C source code for .

- converting the data to an ASCII (plain text) format (`gshhs.c`);
- thinning the data by reducing the number of points sensibly (`gshhs_dp.c`).

Their thinning software uses an algorithm devised by Douglas and Peucker (1973), whose initials `dp` appear in the file name. The `dp` is also an abbreviation of "decimate polygons".

We have created a function called `importGSHHS` that works directly on a specified binary data file from Wessel (resolution choice left to the user) to create a `PBSmapping` PolySet. The user can choose to further alter the resolution of the newly created PolySet using our function `thinPolys`. Alternatively, the user can thin Wessel's full-resolution database (`gshhs_f.b`) directly using `gshhs_dp.c` (after compilation to an executable file) to a desired resolution, then use `PBSmapping`'s `importGSHHS` on the modified binary database. At the time of writing, `importGSHHS` supports Wessel's format for data files version 2.2.0, created July 15,

2011. Wessel's database `gshhs+wdbii_2.2.0.zip` contains geographical coordinates for shorelines (`gshhs`), rivers (`wbd_rivers`), and borders (`wdb_borders`). The latter two come from World DataBank II (WDBII) with the five resolutions mentioned above.

`PBSmapping` includes four data sets derived from the GSHHS databases (Table 2). These all use longitude-latitude (`LL`) coordinates. The `nepac` data sets contain the northeastern Pacific Ocean shoreline in a region that extends roughly from California to Alaska (Figure 2), and the `world` data sets cover the planet (Figure 1). As discussed in section 2.2, longitude coordinates x take continuous values meaningful for the intended map, with $x = 0°$ on the Greenwich prime meridian.

Table 2. PolySets derived from various resolution GSHHS databases.

PolySet	Wessel DB	Thin	Longitude	Latitude	Vertices	Polygons
`nepacLL`*	`gshhs_h.b`	0.2 km	$-190° \le x \le -110°$	$34° \le y \le 72°$	75,305	495
`nepacLLhigh`	`gshhs_f.b`	0.1 km	$-190° \le x \le -110°$	$34° \le y \le 72°$	192,762	9,986
`worldLL`*	`gshhs_l.b`	5.0 km	$-20° \le x \le 360°$	$-90° \le y \le 84°$	30,129	190
`worldLLhigh`*	`gshhs_i.b`	1.0 km	$-20° \le x \le 360°$	$-90° \le y \le 84°$	187,101	1,367

*Excludes polygons with fewer than 15 vertices after thinning.

Explicitly, the commands to create the above PolySets are:

```
worldLL <-importGSHHS("gshhs_l.b", xlim=c(-20,360), ylim=c(-90,90), level=1,
      n=15, xoff=0)
worldLL <- .fixGSHHSWorld(worldLL)

worldLLhigh <-importGSHHS("gshhs_i.b", xlim=c(-20,360), ylim=c(-90,90),
      level=1, n=15, xoff=0)
worldLLhigh <- .fixGSHHSWorld(worldLLhigh)

nepacLL <-importGSHHS("gshhs_h.b", xlim=c(-190,-110), ylim=c(34,72), level=1,
      n=15, xoff=-360)

nepacLLhigh <-importGSHHS("gshhs_f.b", xlim=c(-190,-110), ylim=c(34,72),
      level=1, n=0, xoff=-360)
nepacLLhigh <- thinPolys(nepacLLhigh, tol=0.1, filter=3)
```

2.5. Bathymetry Data

Smith and Sandwell (1997) have produced global seafloor topography from satellite altimetry and ship depth soundings. Their database appears on the Internet at http://topex.ucsd.edu/cgi-bin/get_data.cgi. A web-based data acquisition form allows users to extract a region after entering longitude and latitude coordinate ranges. Appendix B documents how to import their data for use with `PBSmapping`.

R provides a `contour` function to plot contour lines. This function lacks a `save` argument and does not return contour coordinates. Instead, the `contourLines` function accomplishes this task, giving a list that captures continuous contours as single polylines (Figure 6).

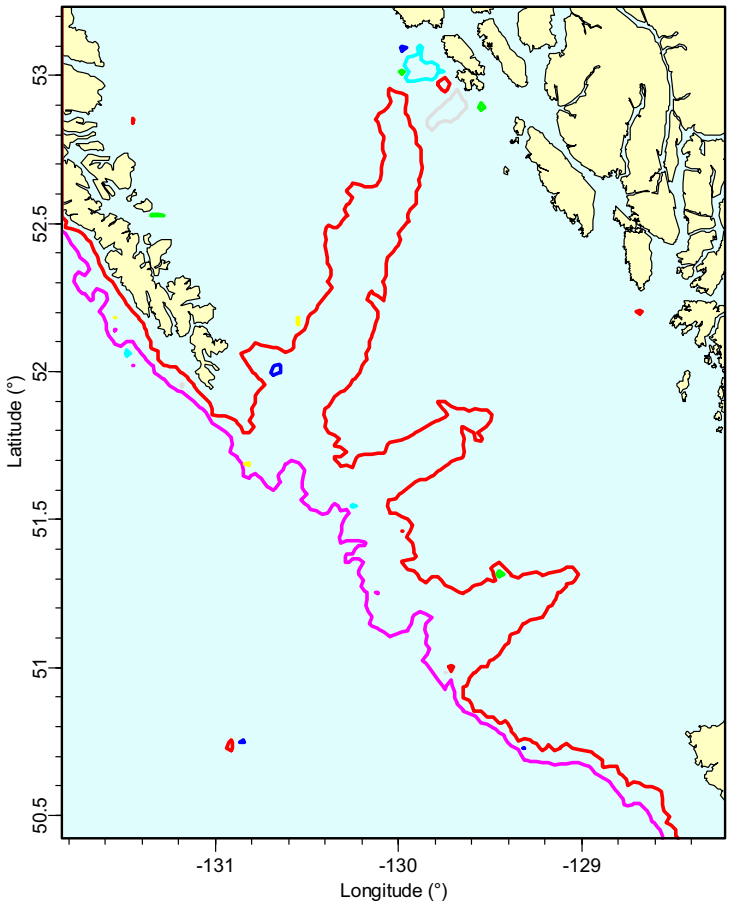

Figure 6. The R `contourLines` function returns a single polyline for each continuous contour.

Our function `convCP` converts the list output from `contourLines` into a list object that has two components: a PolySet with contour coordinates and PolyData with the depth of each contour. The package `PBSdata` includes a data set (`isobaths`) of bathymetric contours for Canada's Pacific coast. In addition, several functions ease the manual procedure of converting polylines into polygons, including

- `convLP` to convert two polylines into a single polygon;
- `closePolys` to close the polygons in a PolySet;
- `fixBound` to fix the boundary points of a PolySet.

2.6. Examples and Applications

Our library includes an illustrative PolySet `towTracks` containing the longitude-latitude coordinates of 45 tow tracks from a longspine thornyhead (*Sebastolobus altivelis*) survey in

2001. Figure 7 portrays these data relative to the west coast of Vancouver Island, drawn with shoreline data clipped from the PolySet `nepacLL`. The PolyData object `towData` specifies the depth of each tow, represented in the figure by colours corresponding to depth intervals (black = 500-800 m, red = 800-1200 m, dark blue = 1200-1600 m).

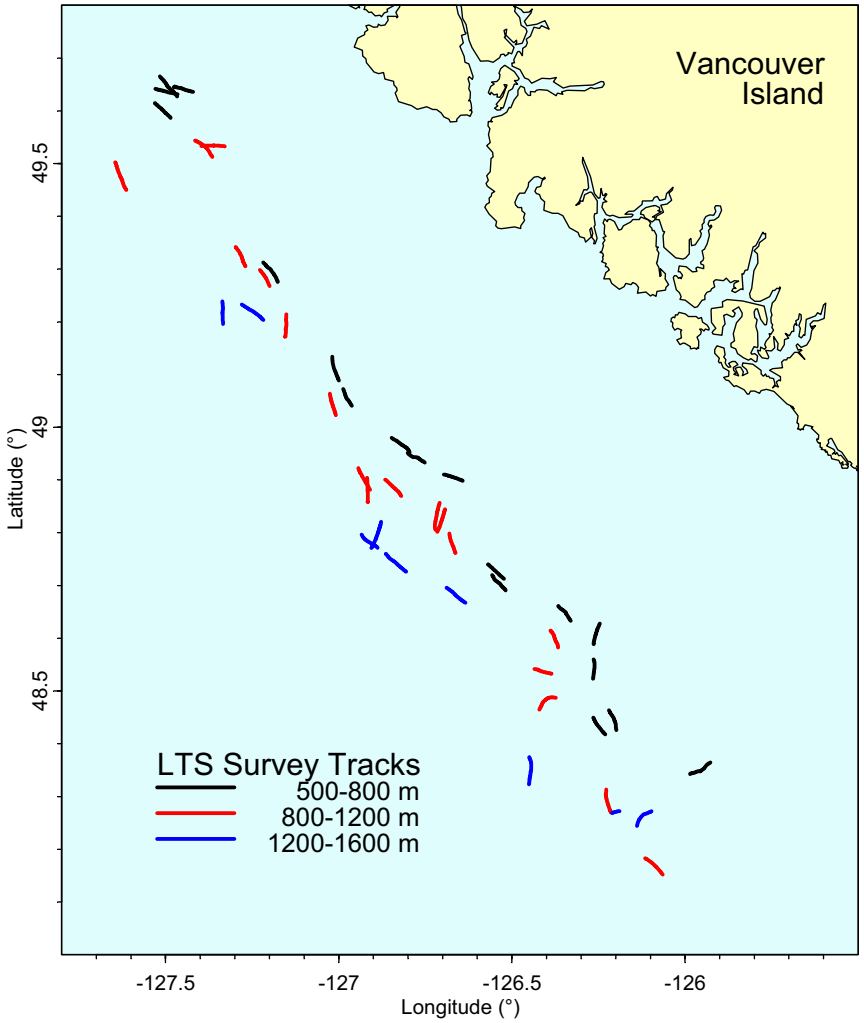

Figure 7. Tracks for 45 tows performed during the 2001 longspine thornyhead (*Sebastolobus altivelis*) survey along the west coast of Vancouver Island (Starr et al. 2002). Each tow track is colour-coded by depth stratum. Data: PolySet `towTracks` and PolyData `towData`.

Figure 8 illustrates the use of our software to calculate polygon areas using the function `calcArea`. We examine a region along the south-west British Columbia coast that includes a cluster of islands in the Strait of Georgia. Shoreline data come from the PolySet `nepacLLhigh`. Because area calculations do not make sense in the longitude-latitude projection, we convert the PolySet to UTM coordinates, with comparable `X` and `Y` coordinates (km), and then clip to the desired region. (The `calcArea` function will also automatically convert PolySets with `projection="LL"` to UTM before calculation.) The figure shows areas for six selected islands, highlighted in yellow. Island centroids, derived using `calcCentroid`, give reference coordinates for printing island names and areas.

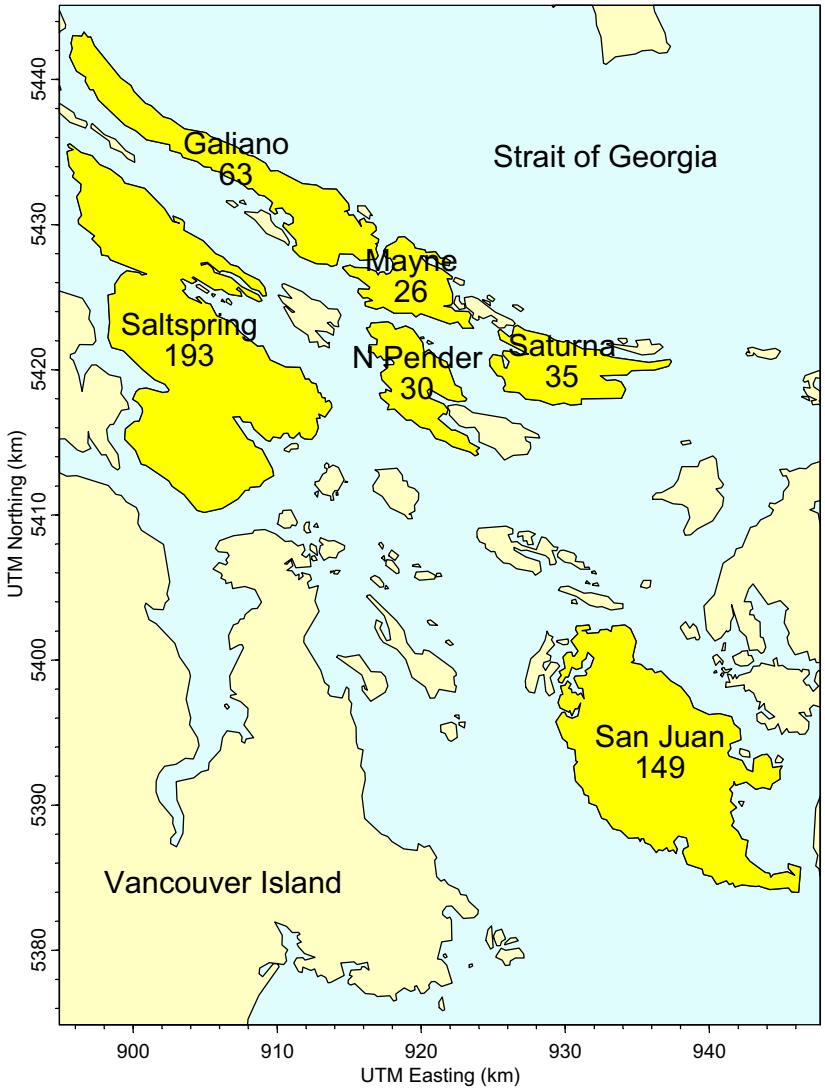

Figure 8. Areas (km^2) of selected islands in the southern Strait of Georgia. Shoreline data have been clipped from `nepacLLhigh` after conversion to UTM coordinates.

Figure 9 portrays data from Pacific ocean perch (*Sebastes alutus*) surveys conducted along the central BC coast during the years 1966-1989. The EventData object `surveyData` contains information from each tow, including the longitude, latitude, depth, catch, and effort (tow duration). These data also imply the computed value of catch per unit effort (CPUE = catch/effort). Code for this figure includes the following key function calls:

- `plotMap` to draw a coastal map of this region, clipped from `nepacLL`;
- `makeGrid` to create a grid in the region of interest;
- `findCells` to associate tows with the appropriate grid cells;
- `combineEvents` to calculate the mean CPUE within each cell;
- `addPolys` to draw cells with colours (in the `polyProps` argument) scaled to the CPUE;
- `points` (the native R function) to plot events on the map.

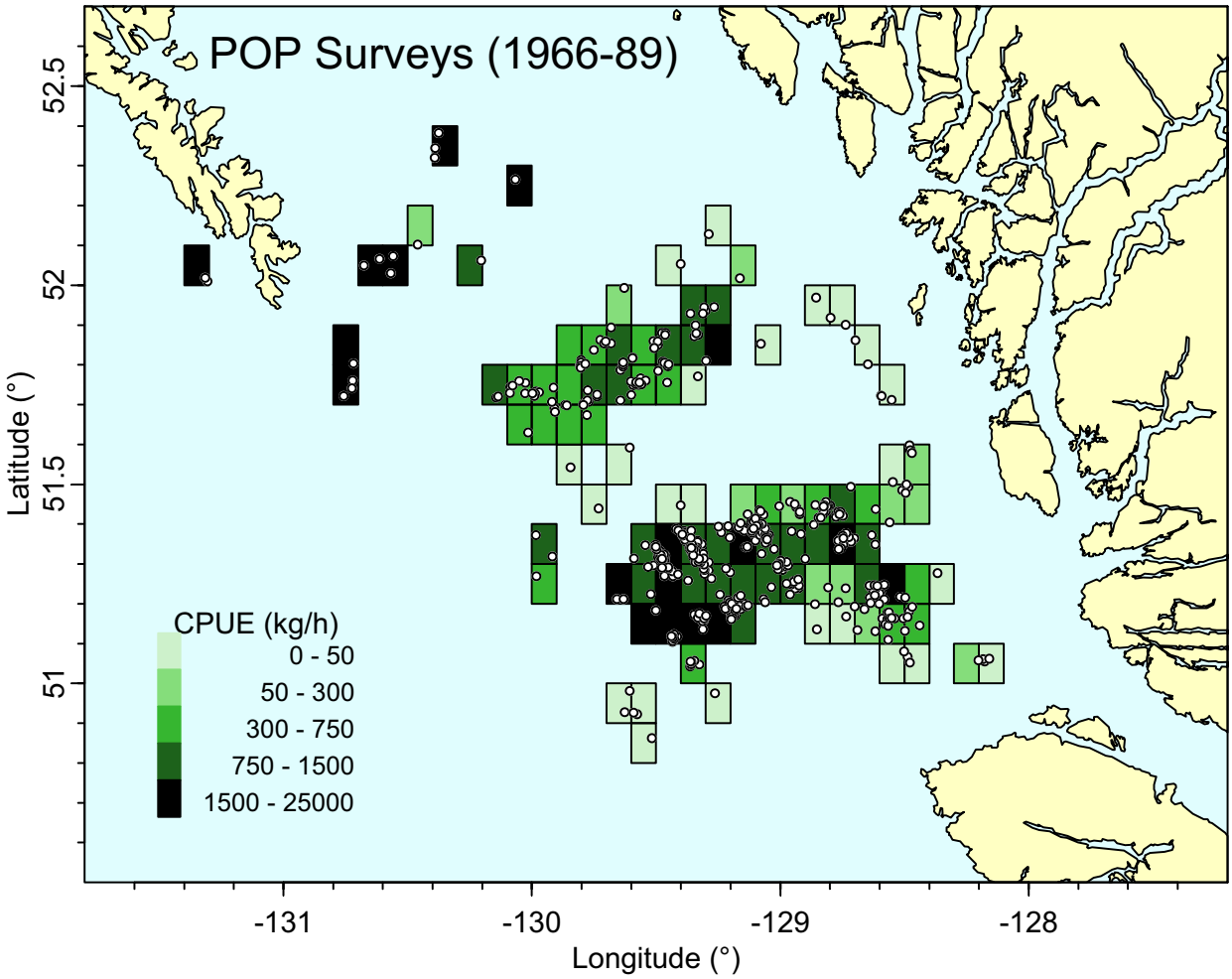

Figure 9. Portrayal of `surveyData` from Pacific ocean perch (*Sebastes alutus*) surveys in the central coast region of British Columbia from 1966-89, with shoreline data clipped from `nepacLL`. Colours portray the mean catch per unit effort (CPUE) within each grid cell (0.1º by 0.1º). Circles show locations of individual tows.

`PBSmapping` can also display non-geographical data, such as technical drawings, network diagrams, and transportation schematics. For example, we use a PolySet to construct the proof of Pythagoras' Theorem in Figure 10, where the caption explains the logic leading to the famous result $a^2 + b^2 = c^2$. Incidentally, Devlin (1998, chapter 6, p. 221) mentions an historical incident that nicely distinguishes maps from network diagrams. A now familiar drawing of the London Underground (see the PDF file marked "Standard Tube map" at the web site http://www.tfl.gov.uk/gettingaround/1106.aspx) fails to represent geography correctly, but contains exactly the information passengers need to navigate the system. It took two years for the designer, Henry C. Beck, to persuade his superiors that his drawing would prove useful to the public.

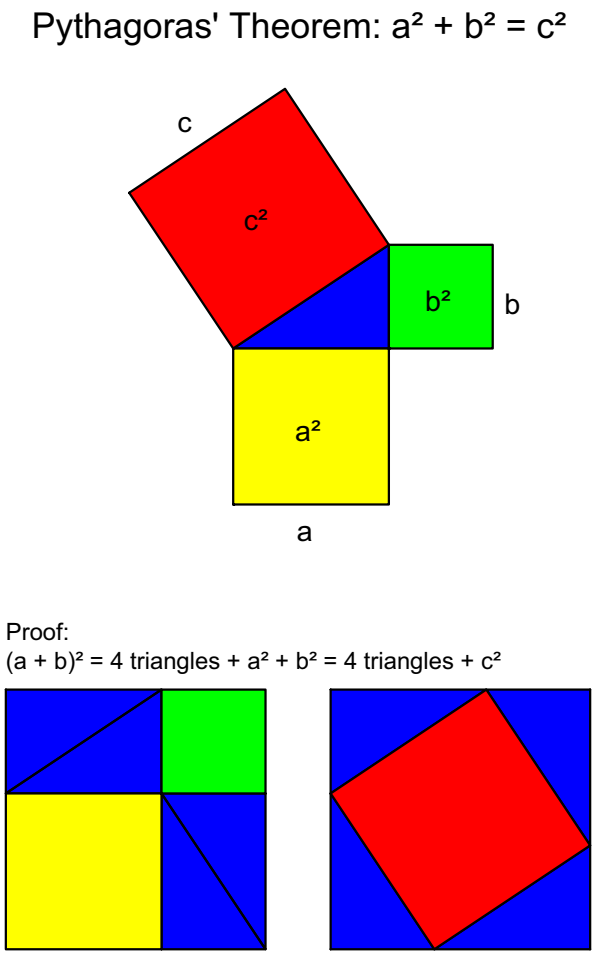

Figure 10. Proof of Pythagoras' Theorem. A PolySet defines all geometric objects in this figure, and PolyData determine the colours for plotting. Four blue triangles plus the yellow square (a^2) and the green square (b^2) equal four blue triangles plus the red square (c^2); consequently, $a^2 + b^2 = c^2$.

2.7. Strengths, Limitations, and Alternatives

`PBSmapping` works with data exported from database tables, where records may not have a definite order. The `POS` field in our PolySet definition imposes the required order for polylines and polygons. This field also provides a convenient means of distinguishing inner and outer boundaries. Our PolySets have a flat structure with at most two levels, corresponding to primary and secondary IDs. We have found these limitations acceptable in the context of our work. Sceptical readers might challenge our choices and prefer more complex hierarchical structures. For example, Becker and Wilks (1993, 1995) define polygons as composites of polylines, so that a common boundary between two regions need be defined only once and then referenced in each regional definition. In our approach, all vertices of a common boundary must be repeated in each regional definition.

We designed our software explicitly to address a few key issues in the spatial representation of fishery data:

- easy importation from databases, Geographic Information Systems, and other sources, such as the shoreline data compiled by Wessel and Smith (1996);
- precise control over the boundaries chosen for clipping from a larger map;
- support for longitude-latitude and UTM easting-northing coordinates;
- computational ability to associate events with polygons in which they lie;
- flexible plotting tools that summarise events within grids and other polygons.

Different purposes could well lead to other designs.

In addition to their comprehensive shoreline database, Wessel and Smith have designed and released a free collection of Generic Mapping Tools (GMT; http://gmt.soest.hawaii.edu/) that provide a serious alternative to our software. These tools operate in the DOS/UNIX environment and support many more projections than `PBSmapping`. They also store polygons in a more efficient file format than our PolySet data frames. We designed `PBSmapping` for the R environment, with its rich support for statistical and mathematical analysis. We have also included numerous algorithms from computational geometry, such as `findPolys` and `joinPolys`. Readers may, however, find GMT more useful for map formats not supported in `PBSmapping`. Appendix C shows some comparative examples of code written in both environments.

Because `PBSmapping` includes features often supported by a Geographic Information System (GIS), a free GIS package might also provide an alternative to the software described here. The FreeGIS web site (http://www.freegis.org) summarizes the current status of free GIS programs and data. Their listings receive frequent updates and show a pattern of steady growth.

3. COMMAND-LINE UTILITIES

The `PBSmapping` package for R includes several algorithms that we have also implemented as stand-alone command-line utilities. These can handle very large data sets that may be too large for the R working environment. Furthermore, some users may wish to implement computational geometry calculations without reference to the R language. Our utilities make this possible by directly processing text files with the appropriate data format. They have been compiled with the same C code used for the dynamically linked library (DLL) in R. For each utility, a corresponding `.c` file provides a front end to shared code for the algorithms. Source code appears in the R library directory `\PBSmapping\Utils\`.

3.1. `clipPolys.exe` (Clip Polygons)

The application `clipPolys.exe` reads an ASCII file containing a PolySet (explained further below) and then clips it. The command

`clipPolys.exe /i` *IFILE* `[/o` *OFILE*`] [/x` *MIN_X*`] [/X` *MAX_X*`] [/y` *MIN_Y*`] [/Y` *MAX_Y*`]`

has five arguments as follows:

- `/i` *IFILE* ASCII input file containing a PolySet (required);
- `/o` *OFILE* ASCII output file (defaults to standard output);
- `/x` *MIN_X* lower `X` limit (defaults to minimum `X` in the PolySet);
- `/X` *MAX_X* upper `X` limit (defaults to maximum `X` in the PolySet);
- `/y` *MIN_Y* lower `Y` limit (defaults to minimum `Y` in the PolySet);
- `/Y` *MAX_Y* upper `Y` limit (defaults to maximum `Y` in the PolySet).

The first line of the PolySet input file must contain the field names (`PID`, `SID`, `POS`, `X`, `Y`), where `SID` is optional. Subsequent lines must contain the data, with the same number of fields per row as in the header line. All fields must be delimited by white space. The program generates a properly formatted PolySet. By default (unless otherwise specified by `/o`), this result goes to standard output, which can be redirected to a text file (e.g., `> file.txt`).

3.2. `convUL.exe` (Convert between UTM and LL)

The application `convUL.exe` reads an ASCII file containing two fields named `X` and `Y`, as described further below. The command

`convUL.exe /i` *IFILE* `[/o` *OFILE*`] (/u | /l) [/m] /z` *ZONE*

has the arguments:

- `/i` *IFILE* ASCII input file containing the `X` and `Y` data (required);
- `/o` *OFILE* ASCII output file (defaults to standard output);
- `/u` (or `/l`) convert to UTM (longitude-latitude) coordinates (required);

- `/m` use metres instead of kilometres as UTM measurement;
- `/z` *ZONE* source or destination zone for the UTM coordinates (required).

The input file must have an initial header line with field names, including `X` and `Y`. Subsequent lines contain the data, with all fields separated by white space. The program converts each (`X`, `Y`) pair to a new pair (`X2`, `Y2`). The output file matches the input file, with the fields (`X2`, `Y2`) appended to the end of each line. The default standard output can be redirected to a text file.

3.3. `findPolys.exe` (Points-in-Polygons)

The application `findPolys.exe` reads two ASCII files: one containing a PolySet and the other containing EventData. The program then determines which events fall inside the available polygons. The command

`findPolys.exe /p` *POLY_FILE* `/e` *EVENT_FILE* `[/o` *OFILE*`]`

has the arguments:

- `/p` *POLY_FILE* ASCII input file containing the PolySet (required);
- `/e` *EVENT_FILE* ASCII input file containing EventData (required);
- `/o` *OFILE* ASCII output file (defaults to standard output).

The header line in both input files must contain field names, and subsequent lines must contain the relevant fields of data delimited by white space. The PolySet must have field names (`PID`, `SID`, `POS`, `X`, `Y`), where `SID` is optional. The EventData must have fields (`EID`, `X`, `Y`). The program writes a properly formatted LocationSet with three or four columns (`EID`, `PID`, `SID`, `Bdry`), where `SID` may be missing (Section 2.1). The default standard output can be redirected to a text file.

ACKNOWLEDGEMENTS

We thank Dr. Jim Uhl and Dr. Peter Walsh in the Computing Science Department, Malaspina University-College, for encouraging and facilitating the role of students in applied fisheries research. Without the dedicated work of these students, named in the Preface, we could not have produced the software described here. We also acknowledge the valuable shoreline and bathymetry databases compiled by Dr. Paul Wessel, Dr. Walter Smith, and Dr. D. T. Sandwell (Wessel and Smith 1996; Smith and Sandwell 1997). In particular, we thank Dr. Paul Wessel for permission to redistribute data from the GSHHS database. Code from other authors seriously enhances this version of `PBSmapping`. Dr. Alan Murta (through Toby Howard) has generously given us permission to use his General Polygon Clipper (GPC) library © Copyright 1997-2012, Advanced Interfaces Group, University of Manchester, implemented in our `joinPolys` function. Similarly, Dr. Gary Robinson has kindly allowed us to use his code for a stack-based Douglas-Peuker line simplification routine, implemented in our `thinPolys` function. Our colleague Brian Krishka helped prepare various data objects. The `PBSmapping` package could not exist without R and GCC. We express admiration and gratitude to the remarkable teams that build, document, and distribute such outstanding free software.

REFERENCES

Anonymous. 1998. The ellipsoid and the Transverse Mercator projection. Geodetic Information Paper No. 1 (version 2.2). Ordnance Survey, Southampton, UK. 20 p. URL: http://www.ordsvy.gov.uk/.

Becker, R.A., Chambers, J.M., and Wilks, A.R. 1988. The new S language: a programming environment for data analysis and graphics. Wadsworth and Books/Cole. Pacific Grove, CA.

Becker, R.A., and Wilks, A.R. 1993. Maps in S. Statistics Research Report 93.2. AT&T Bell Laboratories, Murray Hill, NJ. 21 p. URL: http://www.research.att.com/areas/stat/doc/.

Becker, R.A., and Wilks, A.R. 1995 (rev. 1997). Constructing a geographical database. Statistics Research Report 95.2. AT&T Bell Laboratories, Murray Hill, NJ. 23 p. URL: http://www.research.att.com/areas/stat/doc/.

Boers, N.M., Haigh, R., and Schnute, J.T. 2004. PBS Mapping 2: developer's guide. Canadian Technical Report of Fisheries and Aquatic Sciences 2550.

Bourke, P. 1988 Jul. Calculating the area and centroid of a polygon. URL: http://astronomy.swin.edu.au/~pbourke/geometry/polyarea/ Accessed Aug. 3, 2004.

Chamberlain, R. 2001 Feb. Q5.1: what is the best way to calculate the distance between 2 points. URL: http://www.census.gov/cgi-bin/geo/gisfaq?Q5.1 Accessed Aug. 3, 2004.

de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. 2000. Computational geometry: algorithms and applications: second edition. Springer: Berlin.

Devlin, K.J. 1998. The language of mathematics: making the invisible visible. W. H. Freeman and Company. New York, NY. 344 p. (Reference taken from the first paperback printing 2000)

Douglas, D.H., and Peucker, T.K. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Canadian Cartographer 10:112-22.

Environmental Systems Research Institute (ESRI). 1996. ArcView GIS: the geographic information system for everyone. ESRI Press, Redlands, CA.

Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F. 1996. Computer graphics principles and practice: second edition in C. Addison-Wesley Publishing Co. Boston, MA.

Haigh, R., and Schnute, J. 1999. A relational database for climatological data. Canadian Manuscript Report of Fisheries and Aquatic Sciences 2472. 26 p.

Hains, E. 1994. Point in polygon strategies. Chapter 1.4, p. 24-46 *in*: Heckbert, P.S. 1994. Graphics Gems IV. Academic Press, San Diego, CA. 575 p.

Murta, A. 2004 Jul 15. General polygon clipper homepage. URL: http://www.cs.man.ac.uk/~toby/alan/software/ Accessed Aug. 3, 2004.

Ordnance Survey. (2010) A guide to coordinate systems in Great Britain. Report D00659 (v2.1). Southampton, UK.
URL: http://www.ordnancesurvey.co.uk/oswebsite/gps/docs/A_Guide_to_Coordinate_Systems_in_Great_Britain.pdf.

Rokne, J. 1996. The area of a simple polygon. p. 5-6 *in*: Arvo, J. 1996. Graphics Gems II. Academic Press. San Diego, CA. 672 p.

Rutherford, K.L. 1999. A brief history GFCATCH (1954-1995), the groundfish catch and effort database at the Pacific Biological Station. Canadian Technical Report of Fisheries and Aquatic Sciences 2299. 66 p.

Schnute, J.T., Boers, N.M., and Haigh, R. 2003. PBS Software: maps, spatial analysis, and other utilities. Canadian Technical Report of Fisheries and Aquatic Sciences 2496: 82 p.

Schnute, J.T., Couture-Beil, A., and Haigh, R. 2006. PBS Modelling 1: user's guide. Canadian Technical Report of Fisheries and Aquatic Sciences 2674: viii + 114 p.

Schnute, J.T., Haigh, R., Krishka, B.A., and Starr, P. 2001. Pacific ocean perch assessment for the west coast of Canada in 2001. Canadian Science Advisory Secretariat (CSAS) Research Document 2001/138. 90 p.

Schnute, J.T., Wallace, C.G., and Boxwell, T.A. 1996. A relational database shell for marked Pacific salmonid data (Revision 1). Canadian Technical Report of Fisheries and Aquatic Sciences 2090A. 28 p.

Sinclair, C.A., and Olsen N. 2002. Groundfish research cruises conducted by the Pacific Biological Station, Fisheries and Oceans Canada, 1944-2002. Canadian Manuscript Report of Fisheries and Aquatic Sciences 2617. 91 p.

Sipser, M. 1997. Introduction to the theory of computation. PWS Publishing Company. Boston, MA. 396 p.

Smith, W.H.F., and Sandwell, D.T. 1997. Global seafloor topography from satellite altimetry and ship depth soundings. Science 277:1957-1962.

Starr, P.J., Krishka, B.A., and Choromanski, E.M. 2002. Trawl survey for thornyhead biomass estimation off the west coast of Vancouver Island, September 15 – October 2, 2001. Canadian Technical Report of Fisheries and Aquatic Sciences 2421. 60 p.

Venables, W.N., and Ripley, B.D. 1999. Modern applied statistics with S-PLUS (3rd Edition). Springer-Verlag. New York, NY. 501 p.

Venables, W.N., and Ripley, B.D. 2000. S programming. Springer-Verlag. New York, 264 p.

Wessel, P., and Smith, W.H.F. 1996. A global, self-consistent, hierarchical, high-resolution shoreline database. Journal of Geophysical Research 101:8741-8743.
URL: http://www.soest.hawaii.edu/pwessel/pwessel_pubs.html.

Wikipedia. 2004. Earth radius. URL: http://en.wikipedia.org/wiki/Earth_radius Accessed Aug. 19, 2004.

Wirth, N. 1975. Algorithms + data structures = programs. Prentice-Hall. Englewood Cliffs, NJ. 366 p.

Appendix A. PBSdata package

This appendix documents the objects available in the R-package `PBSdata`, which is not distributed on CRAN but remains available on Google Code: http://code.google.com/p/pbs-data/. Fisheries and Oceans personnel can also obtain the package from the PBS Intranet website: http://svbcpbsgfiis/sql/. Look for a link on the left entitled "Most recent PBS R Packages".

Table A1. Data sets available in `PBSdata`.

Object	Description
bctopo	Topo: British Columbia Sea Floor Topography
bgcp	Topo: Biogeochemical Provinces
claradat	Data: Tow Catches of Species in Queen Charlotte Sound
dbr.rem	Data: Annual Catches of Rockfish by Sector
eez.bc	Topo: Exclusive Economic Zone for BC Coast
fos.fid	Code: Fishery Codes in GFFOS
gear	Code: Gear Codes for Various DFO Databases
hsgrid	Topo: Hecate Strait Assemblage Survey Grid
hsisob	Topo: Hecate Strait Isobaths
hssa	Topo: Hecate Strait Survey Area
iphc.rbr	Data: Longline Indices of Rockfish Catch from the IPHC SSA
iphc.rer	Data: Longline Indices of Rockfish Catch from the IPHC SSA
iphc.yyr	Data: Longline Indices of Rockfish Catch from the IPHC SSA
isobath	Topo: Isobaths (100 to 1800 m, at 100 m intervals)
locality	Topo: Localities in Pacific Marine Fisheries Commission Minor Areas
ltea	Topo: Longspine Thornyhead Exploratory Management Areas
ltmose07	Topo: Longspine Thornyhead Fishing Grounds (WCVI)
ltmose12	Topo: Longspine Thornyhead Fishing Grounds (WCVI)
ltsa	Topo: Longspine Thornyhead Survey Strata (WCVI)
ltsa.bad	Topo: No-Trawl Zones in Longspine Thornyhead Survey Area
ltxa	Topo: Longspine Thornyhead Experimental Management Areas
major	Topo: Pacific Marine Fisheries Commission Major Areas
minor	Topo: Pacific Marine Fisheries Commission Minor Areas
nage394	Data: Age Frequency by Year for Rougheye Rockfish
orfhistory	Data: Historic Landings of Rockfish in BC
parVec	Data: Initial Parameter Vector for Model Fits
pcoda	Topo: Hecate Strait Pacific Cod Monitoring Survey Areas
pjsa	Code: Paul J Starr Locality Codes
pl230	Topo: 230 Degree True Line from Lookout Island
pmfc	Code: Pacific Marine Fisheries Commission Areas
pop.age	Data: Pacific Ocean Perch Age Data (5AB, 5CD)
pop.pmr.qcss	Data: Pacific Ocean Perch (p, mu, rho) for QCS Synoptic Survey
popa	Topo: Pacific Ocean Perch Population Areas
qcb	Topo: Queen Charlotte Basin Surficial Geology
qcssa	Topo: Queen Charlotte Sound Survey Strata
rca	Topo: Rockfish Conservation Areas

Object	Description
species	Code: Species Codes and Names
spn	Code: Species Code Vector
srfa	Topo: Slope Rockfish Assessment Areas
srfs	Topo: Slope Rockfish Assessment Subareas
testdatC	Data: Fisheries Catch Data with Species by Column
testdatR	Data: Fisheries Catch Data with Species by Row
trawlability	Topo: Fisher Knowledge of Towable Bottom
utilize	Code: Utilization Codes for Various DFO Databases
wchgsa	Topo: West Coast Haida Gwaii Survey Area
wcvisa	Topo: West Coast of Vancouver Island Survey Strata
ymr.rem	Data: Annual Catches of Rockfish by Sector

Appendix B. Bathymetry Data

Smith and Sandwell (1997) have produced a global seafloor topography database from satellite altimetry and ship depth soundings. Using the web-based data acquisition form at http://topex.ucsd.edu/cgi-bin/get_data.cgi, users can extract a region from this database. The form returns an ASCII file containing X, Y, and Z coordinates. To use this data file with `PBSmapping`, first load it into R with the native function `read.table`, which creates a data frame with three fields. Our function `makeTopography` can convert this data frame to a list object with vectors `x` and `y` and an outer product matrix `z`, ready for use by the functions `contour` or `contourLines`. In particular, `contourLines` produces a list object that can be easily converted to a PolySet using `convCP`, which in turn produces a list object consisting of a PolySet (with contour coordinates) and PolyData (with the depth of each contour).

Example

Bathymetry for a small section of the Aleutian Islands, Alaska,
where a user would specify coordinates `xlim=c(-162,-158)` and `ylim=c(53,57)` in the web-based acquisition form referenced above, and save Topography to a file called `aleutian.txt` (also provided in the library directory `PBSmapping\extra\`).

```
require(PBSmapping);
isob <- c(100,500,1000,2500,5000);
icol <- rgb(0,0,seq(255,100,len=length(isob)),max=255);

afile <- paste(system.file(package="PBSmapping"),
         "/extra/aleutian.txt",sep="")
aleutian <- read.table(afile, header=F,col.names=c("x","y","z"))
aleutian$x <- aleutian$x - 360
aleutian$z <- -aleutian$z
alBathy <- makeTopography(aleutian)
alCL <- contourLines(alBathy,levels=isob)
alCP <- convCP(alCL)
alPoly <- alCP$PolySet
attr(alPoly,"projection") <- "LL"

plotMap(alPoly,type="n");
addLines(alPoly,col=icol);
data(nepacLL); addPolys(nepacLL,col="gold");
legend(x="topleft",bty="n",col=icol,lwd=2,legend=as.character(isob));
```

Appendix C. Generic Mapping Tools (GMT)

Generic Mapping Tools (GMT) and `PBSmapping` have many similar features, although they operate in different environments. We built `PBSmapping` for the R statistical platform, whereas Wessel and Smith developed GMT to run as commands for the UNIX operating system. Each environment imposes limitations on its respective tools. The following discussion focuses on image types, one of the fundamental areas where the programs differ.

Images are commonly stored in two basic formats, raster and vector. The raster (or bit map) format uses a grid of squares, where each square is assigned characteristics like colour and transparency. The image's resolution, often measured in "dots per inch", determines the density of the grid. When this density is less than the resolution of the output device, the image may appear jagged because distinct squares are visible. Choosing a sufficiently high-resolution image for an output device may result in a large file size. The vector format stores coordinates for control points of lines, curves, and other shapes. Scaling algorithms use these coordinates to produce an image at any specified size with a consistently smooth appearance. In a mapping context, vector formats are usually preferred over raster formats.

Unlike R, the UNIX environment does not have native support for generating images. Wessel and Smith decided that GMT programs would output (optionally encapsulated) postscript files. This vector-based format is more popular in UNIX than Windows and is poorly supported by some word processors, such as Microsoft Word. On the other hand, `PBSmapping` inherits support from the R environment for common raster (e.g., `BMP`, `JPG`) and vector (e.g., `WMF`) file formats. Users of Windows operating systems may find `PBSmapping`'s output somewhat more convenient than that from GMT.

Converting GMT's postscript output to a better-supported graphics format can be achieved through the Ghostscript graphical user interface GSview (http://www.cs.wisc.edu/~ghost/gsview/). Through an option in GSview's "Edit" menu, the program converts `PS` files to the popular `EMF` and `WMF` vector formats. However, we obtained somewhat erratic results from this process and had greater success with raster images produced with the convert option in the "File" menu.

Figure C1 and Figure C2 compare `PBSmapping` with GMT. We show the code used to produce these images in both environments. Although one R command can span multiple lines, one GMT command cannot. For clarity, however, we span GMT commands across multiple lines in the listing below. In familiar UNIX notation, we indicate spanning by escaping the new-line character with a backslash (\).

Code for Figure C1

R: (Panel A)

```
data(nepacLL);                          # load the nepacLL data set
plotMap(nepacLL,                        # plot the nepacLL data set
      xlim=c(-129.3, -122.2),           # limit the region horizontally
      ylim=c(47.5, 51.5),               # limit the region vertically
      plt=c(0.16, 0.97, 0.16, 0.97),    # specify the plot region size
      col=rgb(255, 255, 195,            # set the foreground colour
              maxColorValue=255),
      bg=rgb(224, 253, 254,             # set the background colour
             maxColorValue=255),
      tck=c(-0.03),                     # set the tick mark length
      cex = 1.8,                        # adjust the font size
      mgp=c(1.9, 0.7, 0));              # adjust the axis label locations
```

GMT: (Panel B)

```
gmtset ANOT_FONT_SIZE = 26p             # set the annotation font size
pscoast -Dh \                           # plot the high resolution data set
    -A0/0/1 \                           # skip inner polygons (holes)
    -R-129.3/-122.2/47.5/51.5 \         # limit the region horizontally and vertically
    -JM7i \                             # use the Mercator projection, 7 inches wide
    -G255/255/195 \                     # set the foreground colour
    -S224/253/254 \                     # set the background colour
    -Ba2/a1WSne \                       # mark every 2 (X) and 1 (Y) degrees on W & S axes
    -W0.5p \                            # set the pen width to 0.5 points
    -P \                                # portrait mode
    > GMT-VI.ps                         # output to the postscript file GMT-VI.ps
```

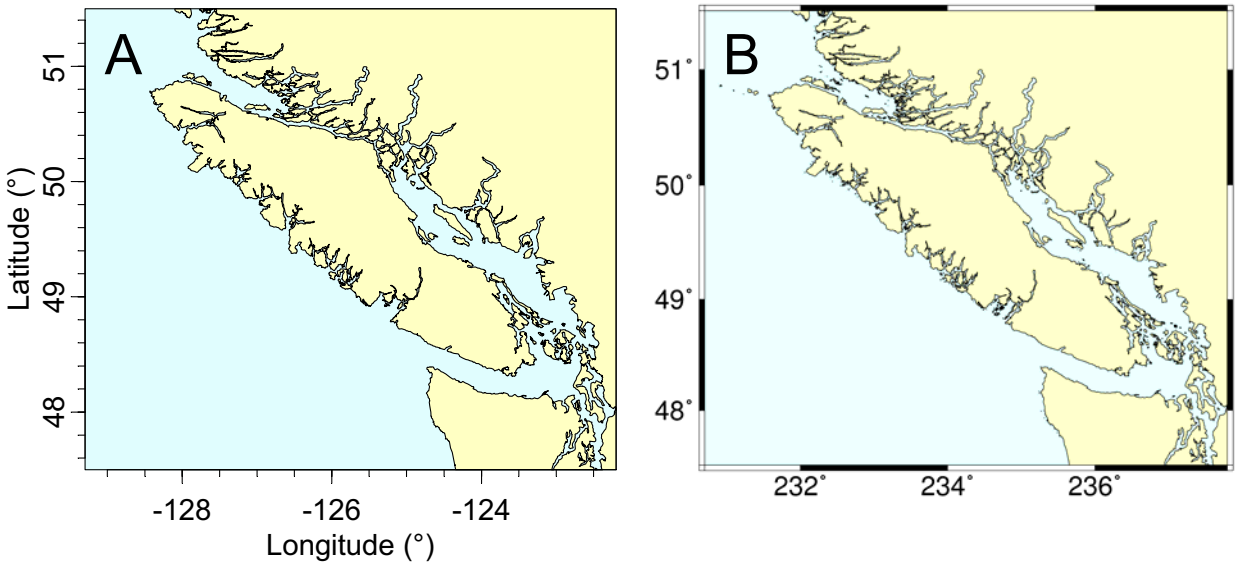

Figure C1. (A) Vancouver Island, as plotted in `PBSmapping`, compared with (B) the same region as output from GMT.

Code for Figure C2

R: (Panel A)

```
data(nepacLL);                          # load the nepacLL data set
plotMap(nepacLL,                        # plot the nepacLL data set
      xlim=c(-127.89, -125.68),         # limit the region horizontally
      ylim=c(47.85, 49.97),             # limit the region vertically
      plt=c(0.16, 0.97, 0.16, 0.97),    # specify the plot region size
      col=rgb(255, 255, 195,            # set the foreground colour
              maxColorValue=255),
      bg=rgb(224, 253, 254,             # set the background colour
              maxColorValue=255),
      tck=c(-0.03),                     # set the tick mark length
      cex=1.8,                          # adjust the font size
      mgp=c(1.9, 0.7, 0));              # adjust the axis label locations
data(towTracks);                        # load the towTracks data set
addLines(towTracks,                     # add the towTracks data set
      col=rgb(255, 0, 0,                # set the colour
              maxColorValue=255),
      lwd=0.5);                         # set the line width
```

GMT: (Panel B)

```
gmtset ANOT_FONT_SIZE = 20p             # set the annotation font size
pscoast -Dh \                           # plot the high resolution data set
      -R-127.89/-125.68/47.85/49.97 \   # limit the region horizontally and vertically
      -JM5i \                           # use the Mercator projection, 5 inches wide
      -G255/255/195 \                   # set the foreground colour
      -S224/253/254 \                   # set the background colour
      -Ba0.5/a0.5WSne \                 # mark every 0.5 (X) and 0.5 (Y) degrees on W & S axes
      -W0.5p \                          # set the pen width to 0.5 points
      -P \                              # portrait mode
      -K \                              # allow for appending more plot code
      > GMT-Tow.ps                      # output to the postscript file GMT-Tow.ps
psxy -R-127.89/-125.68/47.85/49.97 \    # limit the region
      -JM5i \                           # add using the Mercator projection, 5 inches wide
      -W0.5p/255/0/0 \                  # set the pen width to 0.5 points and set the colour
      -M \                              # ASCII file contains multiple polylines
      -H0 \                             # ASCII file does not contain a header
      -O \                              # overlay; lay plot on top of earlier one
      < GMT-Tow.txt \                   # input ASCII file GMT-Tow.txt
      >> GMT-Tow.ps                     # append output to the postscript file GMT-Tow.ps
```

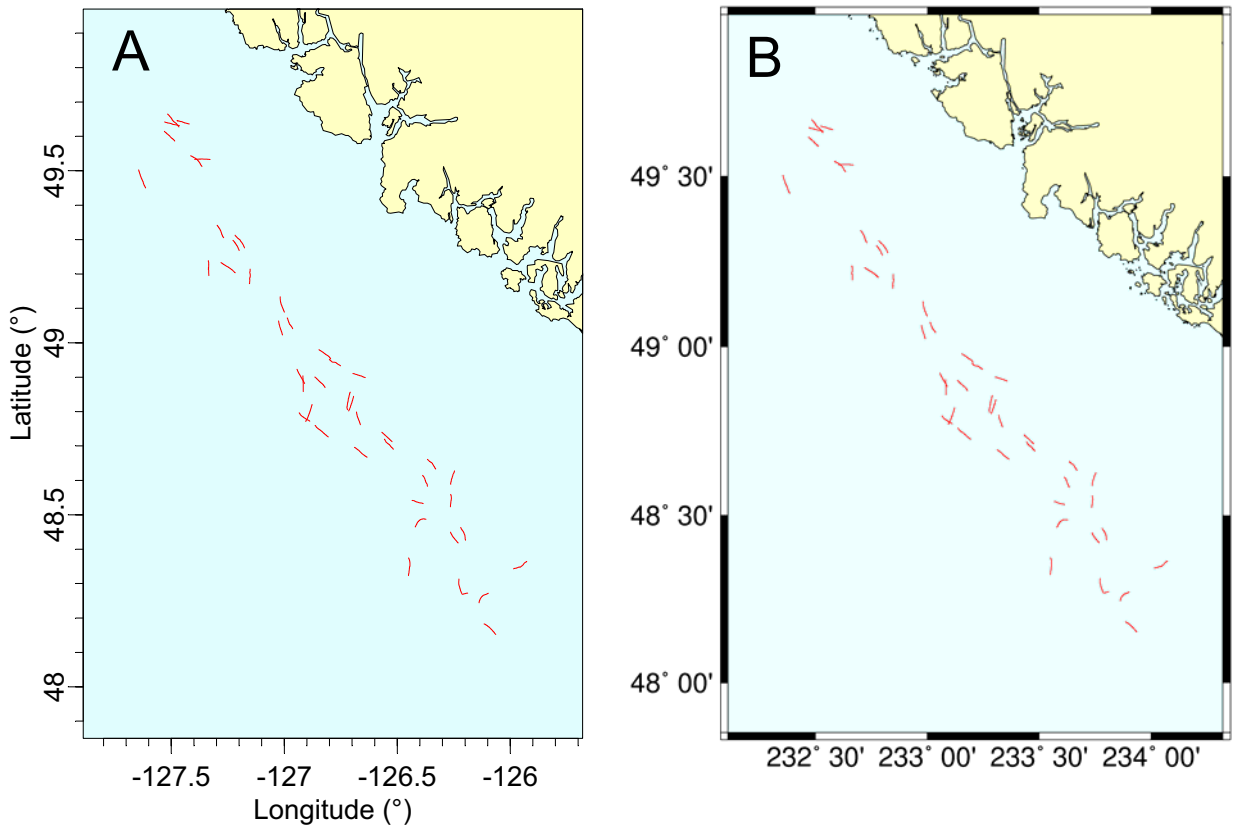

Figure C2. Tow tracks off the west coast of Vancouver Island drawn by (A) `PBSmapping` (B) GMT produced (B).

Format of `GMT-tow.txt`:

```
>                                   # a '>' signifies the start of each polyline
-126.26545 48.523133                # vertices follow: X coordinate, white space, Y coordinate
-126.265233 48.523716
-126.265183 48.524283
…
>
-126.385483 48.532567
-126.3861 48.5327
-126.3868 48.53285
…
```

Appendix D. Source Code for Figures

To help beginners use `PBSmapping`, we include source code for all figures in this report. An intialization function handles most compatibility issues. For example, it creates a global list `PBSval` of colours, dots, and dashes.

Initialization Function

```
.initPBS <- function(new=F) {
#  Sets up colour table and global settings for the demo figures.
#============================================================
   PBSnam <- c("PBSclr","PBSdot","PBSdash")
   PBSclr <- list(black=c(0,0,0),       sea=c(224,253,254),     land=c(255,255,195),
                  red=c(255,0,0),       green=c(0,255,0),       blue=c(0,0,255),
                  yellow=c(255,255,0),  cyan=c(0,255,255),      magenta=c(255,0,255),
                  purple=c(150,0,150),  lettuce=c(205,241,203), moss=c(132,221,124),
                  irish=c(54,182,48),   forest=c(29,98,27),     white=c(255,255,255),
                  fog=c(223,223,223) )
   if (!exists("PBSval") | new==T | (exists("PBSval") &&
all(names(PBSval$PBSclr)!=names(PBSclr))) ) {
      require(PBSmapping)
      PBSclr <- lapply(PBSclr,function(v) {rgb(v[1],v[2],v[3],maxColorValue=255) })
      PBSdot <- 3; PBSdash <- 2
      PBSval <- as.list(PBSnam); names(PBSval) <- PBSnam
      for (i in PBSnam)  PBSval[[i]] <- get(i)
      assign("PBSval", PBSval, pos=1) } }
```

Figure 1 – World UTM Zones

```
.PBSfig01 <- function() { #  World UTM Zones
   .initPBS()
   clr <- PBSval$PBSclr
   data(worldLL);  data(nepacLL)
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #------Plot-the-figure------
   plotMap(worldLL, ylim=c(-90, 90), bg=clr$sea, col=clr$land, tck=-0.023,
           mgp=c(1.9, 0.7, 0), cex=1.2, plt=c(.08,.98,.08,.98))
   # add UTM zone boundaries
   abline(v=seq(-18, 360, by=6), lty=1, col=clr$red)
   # add prime meridian
   abline(v=0, lty=1, lwd=2, col=clr$black)
   # calculate the limits of the 'nepacLL' PolySet
   xlim <- range(nepacLL$X) + 360
   ylim <- range(nepacLL$Y)
   # create and then add the 'nepacLL' rectangle
   region <- data.frame(PID=rep(1,4), POS=1:4, X=c(xlim[1],xlim[2],xlim[2],xlim[1]),
                        Y=c(ylim[1],ylim[1],ylim[2],ylim[2]))
   region <- as.PolySet(region, projection="LL")
   addPolys(region, lwd=2, border=clr$blue, density=0)
   # add labels for some UTM zones
   text(x=seq(183.2, by=6, length=9), y=rep(85,9), adj=0.5, cex=0.65, label=1:9)
   box() }
```

Figure 2 – nepacLL UTM Zones in LL Space

```
.PBSfig02 <- function() {  # nepacLL UTM Zones in LL Space
   .initPBS()
   clr <- PBSval$PBSclr; dot <- PBSval$PBSdot
   data(nepacLL)
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #------Plot-the-figure------
   plotMap(nepacLL, col=clr$land, bg=clr$sea, tck=-0.014,
           mgp=c(1.9,0.7,0), cex=1.2, plt=c(.08,.98,.08,.98))
```

```
   # add lines separating UTM zones
   utms <- seq(-186, -110, 6)
   abline(v=utms, col=clr$red)
   # add the central meridian of zone 6
   abline(v=-147, lty=dot, col=clr$black)
   # create and then add labels for the UTM zones
   cutm <- diff(utms) / 2
   nzon <- length(cutm)
   cutm <- cutm + utms[1:nzon]
   text(cutm,rep(50.75,nzon),c(60,1:(nzon-1)),cex=1.3,col=clr$red)
   box()  }
```

Figure 3 – nepacLL UTM Zones in UTM Space

```
.PBSfig03 <- function() {  # nepacLL UTM Zones in UTM Space
   .initPBS()
   clr <- PBSval$PBSclr; dot <- PBSval$PBSdot
   data(nepacLL)
   zone  <- 6;  xlim  <- range(nepacLL$X);  ylim <- range(nepacLL$Y)
   utms  <- seq(-186,-110,6)  #'utms' vector for creating PolySet and EventData below
   # create UTM zones
   lutms <- data.frame(PID=rep(1:length(utms), each=2),
               POS=rep(c(1,2), times=length(utms)), X=rep(utms,each=2),
               Y = rep(c(ylim[1], ylim[2]), times=length(utms)))
   lutms <- as.PolySet(lutms, projection="LL", zone=zone)
   lutms <- thickenPolys(lutms, tol=25, close=FALSE)
   uutms <- convUL(lutms)
   # create label locations (central meridians)
   lcms  <- data.frame(EID=1:(length(diff(utms)/2)),
               X=utms[1:(length(utms)-1)]+diff(utms)/2,
               Y=rep(50.75, length(diff(utms)/2)))
   lcms  <- as.EventData(lcms, projection="LL", zone=zone)
   ucms  <- convUL(lcms)
   nepacUTM <- nepacLL; attr(nepacUTM,"zone") <- zone  # convert to correct zone
   nepacUTM <- convUL(nepacUTM)
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #------Plot-the-figure------
   plotMap(nepacUTM, col=clr$land, bg=clr$sea, tck=-0.017,
           mgp=c(1.9,0.7,0), cex=1.0, plt=c(0.07,0.97,0.07,0.98))
   addLines(uutms, col=clr$red)
   lines(x=c(500, 500),y=c(4100,7940),lty=dot,col=clr$black)
   text(ucms$X,ucms$Y,c(60,1:(length(utms)-2)),cex=1.3,col=clr$red)
   box()  }
```

Figure 4 – thinPolys on Vancouver Island

```
.PBSfig04 <- function() {  # thinPolys on Vancouver Island
   .initPBS()
   clr <- PBSval$PBSclr;
   data(nepacLL)
   par(mfrow=c(1,2),omi=c(0,0,0,0)) #------Plot-the-figure------
   vi     <- nepacLL[nepacLL$PID==33,]
   xlim   <- range(vi$X) + c(-0.25, 0.25); ylim <- range(vi$Y) + c(-0.25, 0.25)
   # plot left figure (normal Vancouver Island)
   plotMap(vi, xlim, ylim, col=clr$land, bg=clr$sea, tck=-0.028,
           mgp=c(1.9,0.7,0), cex=1.0, plt=c(0.14,1.00,0.07,0.97))
   text(x=xlim[2]-0.5, y=ylim[2]-0.3, "A", cex=1.6)
   # plot right figure (thinned Vancouver Island)
   plotMap(thinPolys(vi, tol=10), xlim, ylim, col=clr$land, bg=clr$sea,
           tck=c(-0.028, 0), tckLab=c(TRUE, FALSE),
           mgp=c(1.9, 0.7, 0), cex=1.0, plt=c(0.00, 0.86, 0.07, 0.97))
   text(x=xlim[2]-0.5, y=ylim[2]-0.3, "B", cex=1.6)
   box() }
```

Figure 5 – joinPolys on Crescents

```
.PBSfig05 <- function() {  # joinPolys on Crescents
   .initPBS(); clr <- PBSval$PBSclr; dash <- PBSval$PBSdash
   radius <- c(5, 4)                 # two radii of the circles
   size   <- abs(diff(radius)) + 0.1 # size of crescent
   shiftB <- 3.5                     # distance to shift second crescent
   pts    <- 120                     # points in outer circle
   cex    <- 1.0                     # character expansion for labels
   off    <- 1.2                     # panel label offset
   xlim   <- c(0, radius[1]*2 + shiftB) + c(-1,1)
   ylim   <- c(0, radius[1]*2) + c(-2,1)
   Mmin   <- .10 # minimum OMI
   Rdin   <- par()$din[2]/par()$din[1]
   Rfig   <- (3*diff(ylim))/(2*diff(xlim))
   if (Rdin > Rfig) {
      width  <- par()$din[1] - 2 * Mmin
      height <- width * (3*diff(ylim))/(2*diff(xlim))
      Mmax   <- (par()$din[2] - height) / 2
      parOmi <- c(Mmax,Mmin,Mmax,Mmin) }
   else {
      height <- par()$din[2] - 2 * Mmin
      width  <- height * (2*diff(xlim))/(3*diff(ylim))
      Mmax   <- (par()$din[1] - width) / 2
      parOmi <- c(Mmin,Mmax,Mmin,Mmax) }
   polyA  <- list()
   for (i in 1:length(radius)) {
      polyA[[i]] <- as.PolySet(data.frame(PID=rep(1,pts), POS = 1:pts,
         X =radius[i]*cos(seq(0, 2*pi, len=pts)),
         Y =radius[i]*sin(seq(0, 2*pi, len=pts))), projection = 1)
      polyA[[i]][, c("X","Y")] <- polyA[[i]][, c("X","Y")] + radius[i] }
   # centre B within A
   polyA[[2]][,c("X","Y")] <- polyA[[2]][,c("X","Y")] + (radius[1]-radius[2])
   # shift B right
   polyA[[2]]$X <- polyA[[2]]$X + size
   # create 'polysA' and 'polysB'
   polyA  <- as.PolySet(joinPolys(polyA[[1]], polyA[[2]], operation="DIFF"), proj=1)
   polyB  <- polyA
   polyB$X<- abs(polyB$X - (radius[1] * 2)) + shiftB
   par(mfrow=c(3,2),mai=c(0,0,0,0),omi=parOmi) #------Plot-the-figure------
   lab    <- list()
   lab$text <- c("Polygon A", "Polygon B", "A \"INT\" B","A \"UNION\" B",
                 "A \"DIFF\" B", "A \"XOR\" B")
   lab$cex <- rep(cex, 6);  lab$x <- rep(mean(xlim), 6);  lab$y <- rep(-0.8, 6)
   # panel A: polyA
   plotMap(polyA,xlim=xlim,ylim=ylim,xlab="",ylab="",axes=F,col=clr$red,plt=NULL)
   text(lab$text[1], x=lab$x[1], y=lab$y[1], cex=lab$cex[1])
   text(xlim[1]+off, ylim[2]-off, "A", cex=1.6);  box()
   # panel B: polyB
   plotMap(polyB,xlim=xlim,ylim=ylim,xlab="",ylab="",axes=F,col=clr$blue,plt=NULL)
   text(lab$text[2], x=lab$x[2], y=lab$y[2], cex=lab$cex[2])
   text(xlim[1]+off, ylim[2]-off, "B", cex=1.6);  box()
   # panels C to F
   ops    <- c(NA, NA, "INT", "UNION", "DIFF", "XOR")
   cols   <- c(NA, NA, clr$red, clr$purple, clr$purple, clr$purple)
   panel  <- c(NA, NA, "C", "D", "E", "F")
   for (i in 3:6) {
      plotMap(NULL,xlim=xlim,ylim=ylim,proj=1,xlab="",ylab="",axes=F,plt=NULL)
      addPolys(polyA, border=clr$red, lty=dash)
      addPolys(polyB, border=clr$blue, lty=dash)
      addPolys(joinPolys(polyA, polyB, operation=ops[i]), col=cols[i])
      text(lab$text[i], x=lab$x[i], y=lab$y[i], cex=lab$cex[i])
      text(xlim[1]+off, ylim[2]-off, panel[i], cex=1.6);  box();  } }
```

Figure 6 – contourLines in Queen Charlotte Sound

```
.PBSfig06 <- function() {  # contourLines in Queen Charlotte Sound
   .initPBS()
   clr <- PBSval$PBSclr;
   data(nepacLL); data(bcBathymetry);
   isob   <- contourLines(bcBathymetry, levels=c(250, 1000))
   p      <- convCP(isob)
   attr(p$PolySet,"projection") <- "LL"
   p$PolyData$col <- rep(c(clr$red, clr$green, clr$blue, clr$yellow,
      clr$cyan, clr$magenta, clr$fog), length=nrow(p$PolyData))
   xlim   <- c(-131.8382, -128.2188)
   ylim   <- c(50.42407, 53.232476)
   region <- clipPolys(nepacLL, xlim=xlim, ylim=ylim)
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #-----Plot-the-figure------
   plotMap(region, xlim=xlim, ylim=ylim, col=clr$land, bg=clr$sea, tck=-0.02,
        mgp=c(2,.75,0), cex=1.2, plt=c(.08,.98,.08,.98))
   addLines(p$PolySet, polyProps=p$PolyData, lwd=3)
   box()  }
```

Figure 7 – towTracks from Longspine Thornyhead Survey

```
.PBSfig07 <- function() {  # towTracks from Longspine Thornyhead Survey
   .initPBS()
   clr <- PBSval$PBSclr;
   data(nepacLL);  data(towTracks);  data(towData);
   # add a colour column 'col' to 'towData'
   pdata  <- towData;  pdata$Z <- pdata$dep
   pdata  <- makeProps(pdata, breaks=c(500,800,1200,1600), "col",
                       c(clr$black, clr$red, clr$blue))
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #------Plot-the-figure------
   plotMap(nepacLL, col=clr$land, bg=clr$sea, xlim=c(-127.8,-125.5), ylim=c(48,49.8),
      tck=-0.01, mgp=c(2,.5,0), cex=1.2, plt=c(.08,1,.08,.98))
   addLines(towTracks, polyProps=pdata, lwd=3)
   # right-justify the legend labels
   temp <- legend(x=-127.6, y=48.4, legend=c(" "," "," "), lwd=3, bty="n",
      text.width=strwidth("1200-1600 m"), col=c(clr$black,clr$red,clr$blue))
   text(temp$rect$left+temp$rect$w, temp$text$y,
      c("500-800 m", "800-1200 m", "1200-1600 m"), pos=2)
   text(temp$rect$left+temp$rect$w/2,temp$rect$top,pos=3,"LTS Survey Tracks");
   text(-125.6,49.7,"Vancouver\nIsland",cex=1.2,adj=1)
   box()  }
```

Figure 8 – calcArea of the Southern Gulf Islands

```
.PBSfig08 <- function() {  # calcArea of the Southern Gulf Islands
   .initPBS()
   clr <- PBSval$PBSclr;
   data (nepacLLhigh)
   xlim   <- c(-123.6, -122.95); ylim <- c(48.4, 49); zone <- 9
   # assign 'nepacLLhigh' to 'nepacUTMhigh' (S62) and change to UTM coordinates
   nepacUTMhigh <- nepacLLhigh;  attr(nepacUTMhigh,"zone" ) <- zone
   nepacUTMhigh  <- convUL(nepacUTMhigh)
   # convert limits to UTM
   temp   <- data.frame(PID=1:4,POS=rep(1,4),X=c(xlim,xlim),Y=c(ylim,rev(ylim)))
   temp   <- convUL(as.PolySet(temp, projection="LL", zone=zone))
   xlim   <- range(temp$X); ylim <- range(temp$Y)
   # prepare areas
   isles  <- clipPolys(nepacUTMhigh,xlim,ylim)
   areas  <- calcArea(isles);
   # PIDs and labels for Gulf Islands
```

```
   bigPID <- areas[rev(order(areas$area)),][c(2:4,6:8),"PID"];
   labelData <- data.frame(PID = bigPID,
      label=c("Saltspring","San Juan","Galiano","Saturna","N Pender","Mayne"))
   labelData <- merge(labelData, areas, all.x=TRUE)
   labelData$label <- paste(as.character(labelData$label),
      round(labelData$area), sep="\n")
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #------Plot-the-figure------
   plotMap(isles, col=clr$land, bg=clr$sea, tck=-.010,
      mgp=c(1.9,.7,0), cex=1, plt=c(.07,.98,.07,.98))
   # add the highlighted Gulf Islands
   bigisles <- isles[is.element(isles$PID,labelData$PID),]
   addPolys(bigisles,col=clr$yellow)
   labXY  <- calcCentroid(isles)
   labXY$Y<- labXY$Y + 2               # centre vertically
   labelData <- merge(labelData, labXY, all.x = TRUE)
   attr(labelData,"projection") <- "UTM"
   addLabels(labelData, placement="DATA", cex=1.25)
   text(898,5385,"Vancouver Island",adj=0, cex=1.25)
   text(925,5435,"Strait of Georgia",adj=0, cex=1.25)  }
```

Figure 9 – combineEvents in Queen Charlotte Sound

```
.PBSfig09 <- function() {  # combineEvents in Queen Charlotte Sound
   .initPBS()
   clr <- PBSval$PBSclr;
   data(nepacLL);  data(surveyData);
   events <- surveyData
   xl     <- c(-131.8, -127.2);  yl <- c(50.5, 52.7)
   # prepare EventData; clip it, omit NA entries, and calculate CPUE
   events <- events[events$X >= xl[1] & events$X <= xl[2] &
                    events$Y >= yl[1] & events$Y <= yl[2], ]
   events <- na.omit(events)
   events$cpue <- events$catch/(events$effort/60)
   # make a grid for the Queen Charlotte Sound
   grid   <- makeGrid(x=seq(-131.6,-127.6,.1), y=seq(50.6,52.6,.1),
                    projection="LL", zone=9)
   # locate EventData in grid
   locData<- findCells(events, grid)
   events$Z <- events$cpue
   pdata  <- combineEvents(events, locData, FUN=mean)
   brks   <- c(0,50,300,750,1500,25000); lbrks <- length(brks)
   cols   <- c(clr$lettuce, clr$moss, clr$irish, clr$forest, clr$black)
   pdata  <- makeProps(pdata, brks, "col", cols)
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #------Plot-the-figure------
   plotMap(nepacLL, col=clr$land, bg=clr$sea, xlim=xl, ylim=yl, tck=-0.015,
           mgp=c(2,.5,0), cex=1.2, plt=c(.08,.98,.08,.98))
   addPolys(grid, polyProps=pdata)
   for (i in 1:nrow(events)) {
      # plot one point at a time for clarity
      points(events$X[i], events$Y[i], pch=16,cex=0.50,col=clr$white)
      points(events$X[i], events$Y[i], pch=1, cex=0.55,col=clr$black) }
   yrtxt  <- paste("(",min(events$year),"-",
                substring(max(events$year),3),")",sep="")
   text(xl[1]+.5,yl[2]-.1,paste("POP Surveys",yrtxt),cex=1.2,adj=0)
   # add a legend; right-justify the legend labels
   temp <- legend(x=xl[1]+.3, y=yl[1]+.7, legend = rep(" ", 5),
              text.width=strwidth("1500 - 25000"), bty="n", fill=cols)
   text(temp$rect$left + temp$rect$w, temp$text$y, pos=2,
        paste(brks[1:(lbrks-1)],brks[2:lbrks], sep=" - "))
   text(temp$rect$left+temp$rect$w/2,temp$rect$top,pos=3,"CPUE (kg/h)",cex=1);  }
```

Figure 10 – Pythagoras' Theorem Visualized

```
.PBSfig10 <- function() {  # Pythagoras' Theorem Visualized
   .initPBS()
   clr <- PBSval$PBSclr;
   data(pythagoras)
   # create properties for colouring the polygons
   pythProps <- data.frame(PID=c(1, 6:13, 4, 15, 3, 5, 2, 14),
                  Z=c(rep(1, 9), rep(2, 2), rep(3, 2), rep(4, 2)))
   pythProps <- makeProps(pythProps, c(0, 1.1, 2.1, 3.1, 4.1), "col",
                  c(clr$blue, clr$red, clr$yellow, clr$green))
   par(mfrow=c(1,1),omi=c(0,0,0,0)) #------Plot-the-figure------
   plotMap(pythagoras, plt=c(.01,.99,.01,.95), lwd=2,
      xlim=c(.09,1.91), ylim=c(0.19,2.86), polyProps=pythProps,
      axes=F, xlab="", ylab="", main="Pythagoras' Theorem: a\262 + b\262 = c\262")
   text(x = 0.1, y = 1.19, adj=0, "Proof:")
   text(x = 0.1, y = 1.10, adj=0,
      "(a + b)\262 = 4 triangles + a\262 + b\262 = 4 triangles + c\262")
   labels <- data.frame(X=c(1.02,1.66,0.65),Y=c(1.50,2.20,2.76),label=c("a","b","c"))
   text(labels$X, labels$Y, as.character(labels$label), cex=1.2)
   text(1.03, 1.81, "a\262", cex=1.2, col=clr$black)
   text(1.43, 2.21, "b\262", cex=1.2, col=clr$black)
   text(0.87, 2.46, "c\262", cex=1.2, col=clr$black)  }
```

Run command file “PBSfigs.r”

```
.PBSfigs <- function(nfigs=1:10) { # Draw all figures with numbers in nfigs
   #while (!is.null(dev.list())) dev.off(dev.cur())
   for (i in nfigs) {
      figStr <- paste(".PBSfig",ifelse(i<10,"0",""),i,sep="")
      get(figStr)();
      cat(figStr); readline(); }  }
```

Appendix E. `PBSmapping` Function Dependencies

This appendix documents function dependencies within `PBSmapping`. All functions appear as underlined entries in the alphabetic list. If a function depends on others, the list of dependencies appears below the underlined name. Following a standard in UNIX and R, functions whose name begins with a period (*dot functions*) are considered hidden from the user, who would normally use only the non-hidden functions that call them. The names here apply primarily to the R working environment, but functions designated '**(C)**' are implemented in C source code and compiled in the DLL for the mapping package. R invokes these functions with the call `.C(…)`. Functions designated '**(S)**' exist as subfunctions only within the R function.

.addAxis

.addBubblesLegend

.addCorners
calcConvexHull

.addFeature
.addProps
.validatePolyProps

.addLabels

.addProps

.calcDist

.calcOrientation
calcOrientation **(C)**

.checkClipLimits

.checkProjection

.checkRDeps

.clip
clip **(C)**
extractPolyData

.closestPoint

.createFastIDdig

.createGridIDs

.createIDs
.createFastIDdig

.expandEdges
.closestPoint
calcConvexHull

.fixGSHHSWorld
findPolys
fixPOS

.getBasename

.getGridPars
makeGrid

.initPlotRegion

.insertNAs

.mat2df

.plotMaps
.addAxis
.addLabels
.initPlotRegion
.validateXYData
addLines
addPoints
addPolys

.preparePolyProps
.createIDs
.validatePolyData

.rollupPolys
rollupPolys **(C)**

.validateData
.createIDs

.validateEventData
.validateData

.validateLocationSet
.validateData

.validatePolyData
.validateData

.validatePolyProps
.validateData

.validatePolySet
.validateData

.validateXYData
.validateData

addBubbles
.addBubblesLegend
.validateEventData

addLabels
.addFeature
.checkProjection
.validateEventData
.validatePolyData
.validatePolySet
calcCentroid
calcMidRange
calcSummary
is.EventData
is.PolyData

addLines
.addProps
.checkProjection
.clip
.createFastIDdig
.createIDs
.preparePolyProps
.validatePolyProps
.validatePolySet
is.PolyData

addPoints
.addFeature
.checkProjection
.validateEventData
.validatePolyData
is.PolyData

addPolys
.addProps
.checkProjection
.clip
.createFastIDdig
.createIDs
.preparePolyProps
.rollupPolys
.validatePolyProps
.validatePolySet
is.PolyData

addStipples
.addFeature
.checkProjection
.clip
.validatePolySet
findPolys
is.PolyData
thickenPolys

appendPolys
.validatePolySet
is.PolySet

as.EventData
.validateEventData
is.EventData

as.LocationSet
.validateLocationSet
is.LocationSet

as.PolyData
.validatePolyData
is.PolyData

as.PolySet
.validatePolySet
is.PolySet

calcArea
.rollupPolys
.validatePolySet
calcArea **(C)**
convUL
is.PolyData

calcCentroid
.rollupPolys
.validatePolySet
calcCentroid **(C)**
is.PolyData

calcConvexHull
.validateXYData
grDevices::chull
is.PolySet

calcLength
.validatePolySet
.rollupPolys
.calcDist
.createIDs

calcMidRange
.validatePolySet
calcSummary
is.PolyData

calcSummary
.createIDs
.rollupPolys
.validatePolySet
is.PolyData

calcVoronoi
.checkRDeps
.validateXYData
deldir::deldir
.addCorners
.expandEdges

clipLines
.clip
.validatePolySet
is.PolySet

clipPolys
.clip
.validatePolySet
is.PolySet

closePolys
.validatePolySet
closePolys **(C)**
is.PolySet

combineEvents
.validateEventData
is.PolyData

combinePolys
.validatePolySet
.createIDs

convCP
is.PolyData

convDP
.validatePolyData
is.PolySet

convLP
.validatePolySet
is.PolySet

convUL
.validateXYData
convUL **(C)**

dividePolys
.validatePolySet
.createIDs

extractPolyData
.createIDs
.validatePolySet
is.PolyData

findCells
.validateEventData
.validatePolySet
findCells **(C)**
is.LocationSet

findPolys
.validateEventData
.validatePolySet
findPolys **(C)**
is.LocationSet

fixBound
.validatePolySet
is.PolySet

fixPOS
.rollupPolys
.validatePolySet
is.PolySet

importEvents
as.EventData

importGSHHS
checkClipLimits
importGSHHS **(C)**

importLocs
as.LocationSet

importPolys
as.PolySet
as.PolyData

importShapefile
.checkRDeps
.getBasename
maptools:Rshapeget **(C)**
.calcOrientation
foreign:read.dbf

is.EventData
.validateEventData

is.LocationSet
.validateLocationSet

is.PolyData
.validatePolyData

is.PolySet
.validatePolySet

isConvex
.validatePolySet
is.PolyData
isConvex **(C)**

isIntersecting
.validatePolySet
is.PolyData
isIntersecting **(C)**

joinPolys
.validatePolySet
is.PolySet
joinPolys **(C)**

locateEvents
is.EventData

locatePolys
.validatePolyData
is.PolySet

makeGrid
is.PolySet

makeProps
.validatePolyData
is.PolyData

makeTopography

outputGSHHS
checkClipLimits
convGSHHS **(C)**

plotLines
.plotMaps
is.PolyData

plotMap
.plotMaps
is.PolyData

plotPoints
.plotMaps
is.PolyData

plotPolys
.plotMaps
is.PolyData

print.EventData
summary.EventData

print.LocationSet
summary.LocationSet

print.PolyData
summary.PolyData

print.PolySet
summary.PolySet

print.summary.PBS

refocusWorld
.createIDs
.shiftRegion **(S)**
.validatePolySet

summary.EventData

summary.LocationSet
.createIDs

summary.PolyData
.createIDs

summary.PolySet
.createIDs

thickenPolys
.calcDist
.createIDs
.validatePolySet
is.PolySet
thickenPolys **(C)**

thinPolys
.validatePolySet
is.PolySet
thinPolys **(C)**

Appendix F. `PBSmapping` Functions and Data

This appendix documents the objects (functions and data) available in `PBSmapping`. Subsequent pages give indexed technical documentation for every object generated from `*.Rd` files written for the R documentation system. The package `PBSmodelling` includes a directory called `PBStools\` that contains useful batch files for building R packages, including the creation of the indexed manual included after Table F1.

Table F1. Functions and data sets in `PBSmapping`, arranged alphabetically within categories.

Category	Object	Description
User constant	`PBSprint`	Specify whether to print summaries
Import functions	`importEvents`	Import a text file and convert into EventData
	`importLocs`	Import a text file and convert into a LocationSet
	`importPolys`	Import a text file and convert into a PolySet
	`importGSHHS`	Import data from a GSHHS database
	`importShapefile`	Import an ESRI shapefile
Plotting functions	`addBubbles`	Add bubbles to maps
	`addLabels`	Add labels to an existing plot
	`addLines`	Add a PolySet to an existing plot as polylines
	`addPoints`	Add EventData/PolyData to an existing plot as points
	`addPolys`	Add a PolySet to an existing plot as polygons
	`addStipples`	Add stipples to an existing plot
	`plotLines`	Plot a PolySet as polylines
	`plotMap`	Plot a PolySet as a map
	`plotPoints`	Plot EventData/PolyData as points
	`plotPolys`	Plot a PolySet as polygons
Computational functions	`appendPolys`	Append a two-column matrix to a PolySet
	`calcArea`	Calculate the areas of polygons
	`calcCentroid`	Calculate the centroids of polygons
	`calcConvexHull`	Calculate the convex hull for a set of points
	`calcLength`	Calculate the length of polylines
	`calcMidRange`	Calculate midpoints of the `X` and `Y` ranges for polygons
	`calcSummary`	Apply functions to polygons in a PolySet
	`calcVoronoi`	Calculate Voronoi tesselation for a set of points
	`clipLines`	Clip a PolySet as polylines
	`clipPolys`	Clip a PolySet as polygons
	`closePolys`	Close a PolySet
	`combineEvents`	Combine measurements of events in same polygon
	`combinePolys`	Combine several polygons into a single polygon
	`convCP`	Convert results from `contourlines` into PolySet
	`convDP`	Convert EventData/PolyData into a PolySet
	`convLP`	Convert polylines into a polygon
	`convUL`	Convert coordinates between UTM/LL projections
	`dividePolys`	Divide a single polygon into several polygons
	`extractPolyData`	Extract PolyData from a PolySet

Category	Object	Description
	findCells	Find cells in a grid that contain events in EventData
	findPolys	Find polygons that contain events in EventData
	fixBound	Fix the boundary points of a PolySet
	fixPOS	Fix the POS column of a PolySet
	isConvex	Determine whether polygons are convex
	isIntersecting	Determine whether polygons are self-intersecting
	joinPolys	Join one or two PolySets using a set theoretic operation
	locateEvents	Locate events on the current plot
	locatePolys	Locate polygons on the current plot
	makeGrid	Make a grid of polygons
	makeProps	Make polygon properties
	makeTopography	Make topography data from freely available online data
	refocusWorld	Refocus the worldLL / worldLLhigh data sets
	thickenPolys	Thicken a PolySet of polygons
	thinPolys	Thin a PolySet of polygons
Object-related functions	as.	Coerce a data frame to an object with class:
	EventData	EventData
	LocationSet	LocationSet
	PolyData	PolyData
	PolySet	PolySet
	is.	Determine whether an object is:
	EventData	EventData
	LocationSet	a LocationSet
	PolyData	PolyData
	PolySet	a PolySet
	print.	Print:
	EventData	an EventData object
	LocationSet	a LocationSet object
	PolyData	a PolyData object
	PolySet	a PolySet object
	summary.PBS	the summary of a PBSmapping object
	summary.	Summarize:
	EventData	EventData
	LocationSet	a LocationSet
	PolyData	PolyData
	PolySet	a PolySet
Data sets	bcBathymetry	Bathymetry data spanning British Columbia's coast
	nepacLL	Northeast Pacific shoreline (normal resolution)
	nepacLLhigh	Northeast Pacific shoreline (high resolution)
	pythagoras	Pythagoras' theorem diagram PolySet
	surveyData	Survey data
	towData	Tow data
	towTracks	Tow track polyline data
	worldLL	World ocean shoreline (normal resolution)
	worldLLhigh	World ocean shoreline (high resolution)

Package 'PBSmapping'

November 22, 2012

Version 2.63

Date 2012-11-19

Title Mapping Fisheries Data and Spatial Analysis Tools

Author Jon T. Schnute ⟨schnutej-dfo@shaw.ca⟩, Nicholas Boers ⟨boersn@macewan.ca⟩, and Rowan Haigh ⟨rowan.haigh@dfo-mpo.gc.ca⟩; GPC library by Alan Murta ⟨gpc@cs.man.ac.uk⟩

Maintainer Jon T. Schnute ⟨schnutej-dfo@shaw.ca⟩

Depends R (>= 2.15.0)

Suggests foreign, maptools, deldir

Enhances gpclib

Description This software has evolved from fisheries research conducted at the Pacific Biological Station (PBS) in Nanaimo, British Columbia, Canada. It extends the R language to include two-dimensional plotting features similar to those commonly available in a Geographic Information System (GIS). Embedded C code speeds algorithms from computational geometry, such as finding polygons that contain specified point events or converting between longitude-latitude and Universal Transverse Mercator (UTM) coordinates. Additionally, we include C code developed by Alan Murta for the General Polygon Clipper (GPC) library (C) Copyright 1997-2012, Advanced Interfaces Group, University of Manchester. PBSmapping also includes data for a global shoreline and other data sets in the public domain. The R directory '.../library/PBSmapping/doc' offers a complete user's guide PBSmapping-UG.pdf, which should be consulted to use all functions in the package effectively.

License file LICENSE

URL http://code.google.com/p/pbs-mapping/, http://code.google.com/p/pbs-mapx/, http://www.cs.man.ac.uk/~toby/gpc/

R topics documented:

addBubbles 47
addLabels 48
addLines 50
addPoints 51
addPolys 52
addStipples 53
appendPolys 54
bcBathymetry 55
calcArea 56

calcCentroid . . . 57
calcConvexHull . . . 57
calcLength . . . 58
calcMidRange . . . 59
calcSummary . . . 60
calcVoronoi . . . 61
clipLines . . . 62
clipPolys . . . 63
closePolys . . . 63
combineEvents . . . 64
combinePolys . . . 65
convCP . . . 66
convDP . . . 67
convLP . . . 68
convUL . . . 69
dividePolys . . . 70
EventData . . . 70
extractPolyData . . . 71
findCells . . . 72
findPolys . . . 73
fixBound . . . 74
fixPOS . . . 75
importEvents . . . 76
importGSHHS . . . 76
importLocs . . . 78
importPolys . . . 78
importShapefile . . . 79
isConvex . . . 80
isIntersecting . . . 81
joinPolys . . . 82
locateEvents . . . 83
locatePolys . . . 84
LocationSet . . . 85
makeGrid . . . 86
makeProps . . . 87
makeTopography . . . 88
nepacLL . . . 89
nepacLLhigh . . . 90
PBSmapping . . . 91
PBSprint . . . 91
placeHoles . . . 92
plotLines . . . 93
plotMap . . . 94
plotPoints . . . 96
plotPolys . . . 98
PolyData . . . 99
PolySet . . . 100
print . . . 102
pythagoras . . . 103
refocusWorld . . . 103
summary . . . 104
surveyData . . . 105
thickenPolys . . . 106
thinPolys . . . 107

towData 108
towTracks 109
worldLL 109
worldLLhigh 110

Index **112**

`addBubbles` *Add Bubbles to Maps*

Description

Add bubbles proportional to some `EventData`'s `Z` column (e.g., catch or effort) to an existing plot, where each unique `EID` describes a bubble.

Usage

```
addBubbles(events, type=c("perceptual","surface","volume"),
   z.max=NULL, max.size=0.8, symbol.zero="+",
   symbol.fg=rgb(0,0,0,0.6), symbol.bg=rgb(0,0,0,0.3),
   legend.pos="bottomleft", legend.breaks=NULL,
   show.actual=FALSE, legend.type=c("nested","horiz","vert"),
   legend.title="Abundance", legend.cex=0.8, ...)
```

Arguments

`events`	EventData to use (*required*).
`type`	scaling option for bubbles where `"perceptual"` emphasizes large z-values, `"volume"` emphasizes small z-values, and `"surface"` lies in between.
`z.max`	maximum value for z (default = `max(events$Z)`); determines the largest bubble; keeps the same legend for different maps.
`max.size`	maximum size (inches) for a bubble representing `z.max`. A legend bubble may exceed this size when `show.actual` is `FALSE` (on account of using `pretty(...)`).
`symbol.zero`	symbol to represent z-values equal to 0.
`symbol.fg`	bubble outline (border) colour.
`symbol.bg`	bubble interior (fill) colour.
`legend.pos`	position for the legend.
`legend.breaks`	break values for categorizing the z-values. The automatic method should work if zeroes are present; otherwise, you can specify your own break values for the legend.
`show.actual`	logical; if `FALSE`, legend values are obtained using `pretty(...)`, and consequently, the largest bubble may be larger than `z.max`. If `TRUE`, the largest bubble in the legend will correspond to `z.max`.
`legend.type`	display format for legend.
`legend.title`	title for legend.
`legend.cex`	size of legend text.
`...`	additional arguments for `points` function that plots zero-value symbols.

Details

Modified from (and for the legend, strongly inspired by) Tanimura et al. (2006) by Denis Chabot to work with **PBSmapping**.

Furthermore, Chabot's modifications make it possible to draw several maps with bubbles that all have the same scale (instead of each bubble plot having a scale that depends on the maximum z-value for that plot). This is done by making `z.max` equal to the largest z-value from all maps that will be plotted.

The user can also add a legend in one of four corners (see `legend`) or at a specific `c(X,Y)` position. If `legend.pos` is `NULL`, no legend is drawn.

Author(s)

Denis Chabot, Maurice Lamontagne Institute, Fisheries and Oceans Canada, Mont-Joli QC

References

Tanimura, S., Kuroiwa, C., and Mizota, T. (2006) Proportional symbol mapping in R. *Journal of Statistical Software* **15**(5).

See Also

`addPolys`, `surveyData`

Examples

```
require(PBSmapping)
data(nepacLL,surveyData)
plotMap(nepacLL, xlim=c(-131.8,-127.2), ylim=c(50.5,52.7),
  col="gainsboro",plt=c(.08,.99,.08,.99))
surveyData$Z <- surveyData$catch
addBubbles(surveyData, symbol.bg=rgb(.9,.5,0,.6),
  legend.type="nested", symbol.zero="+", col="grey")
```

---

addLabels *Add Labels to an Existing Plot*

Description

Add the `label` column of `data` to the existing plot.

Usage

```
addLabels (data, xlim = NULL, ylim = NULL, polyProps = NULL,
           placement = "DATA", polys = NULL, rollup = 3,
           cex = NULL, col = NULL, font = NULL, ...)
```

Arguments

`data`	EventData or PolyData to add (*required*).
`xlim`	range of X-coordinates.
`ylim`	range of Y-coordinates.
`polyProps`	PolyData specifying which labels to plot and their properties. `par` parameters passed as direct arguments supersede these data.

placement	one of "DATA", "CENTROID", "MEAN_RANGE", or "MEAN_XY".
polys	PolySet to use for calculating label placement.
rollup	level of detail at which to process polys, and it should match that in data. 1 = PIDs only, 2 = outer contours only, and 3 = no roll-up.
cex	vector describing character expansion factors (cycled by EID or PID).
col	vector describing colours (cycled by EID or PID).
font	vector describing fonts (cycled by EID or PID).
...	additional par parameters for the text function.

Details

If data is EventData, it must minimally contain the columns EID, X, Y, and label. Since the EID column does not match a column in polys, set placement = "DATA". The function plots each label at its corresponding X/Y coordinate.

If data is PolyData, it must minimally contain the columns PID and label. If it also contains X and Y columns, set placement = "DATA" to plot labels at those coordinates. Otherwise, set placement to one of "CENTROID", "MEAN_RANGE", or "MEAN_XY". When placement != "DATA", supply a PolySet polys. Using this PolySet, the function calculates a centroid, mean range, or mean X/Y coordinate for each polygon, and then links those PolyData with data by PID/SID to determine label coordinates.

If data contains both PID and EID columns, the function assumes it is PolyData and ignores the EID column.

For additional help on the arguments cex, col, and font, please see par.

Value

EventData or PolyData with X and Y columns that can subsequently reproduce the labels on the plot. Modify this data frame to tweak label positions.

See Also

addPoints, calcCentroid, calcMidRange, calcSummary, EventData, plotPoints, PolyData.

Examples

```
#--- create sample PolyData to label Vancouver Island
labelData <- data.frame(PID=33, label="Vancouver Island");
#--- load data
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- plot the map
plotMap (nepacLL, xlim=c(-129, -122.6), ylim=c(48, 51.1))
#--- add the labels
addLabels (labelData, placement="CENTROID", polys=nepacLL, col=2)
```

addLines *Add a PolySet to an Existing Plot as Polylines*

Description

Add a PolySet to an existing plot, where each unique (PID, SID) describes a polyline.

Usage

```
addLines (polys, xlim = NULL, ylim = NULL,
    polyProps = NULL, lty = NULL, col = NULL, ...)
```

Arguments

polys	PolySet to add (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
polyProps	PolyData specifying which polylines to plot and their properties. par parameters passed as direct arguments supersede these data.
lty	vector of line types (cycled by PID).
col	vector of colours (cycled by PID).
...	additional par parameters for the lines function.

Details

The plotting routine does not connect the last vertex of each discrete polyline to the first vertex of that polyline. It clips polys to xlim and ylim before plotting.

For additional help on the arguments lty and col, please see par.

Value

PolyData consisting of the PolyProps used to create the plot.

See Also

calcLength, clipLines, closePolys, convLP, fixBound, fixPOS, locatePolys, plotLines, thinPolys, thickenPolys.

Examples

```
#--- create a PolySet to plot
polys <- data.frame(PID=rep(1,4),POS=1:4,X=c(0,1,1,0),Y=c(0,0,1,1))
polys <- as.PolySet(polys, projection=1)
#--- plot the PolySet
plotLines(polys, xlim=c(-.5,1.5), ylim=c(-.5,1.5), projection=1)
#--- add the PolySet to the plot (in a different style)
addLines(polys, lwd=5, col=3)
```

addPoints *Add EventData/PolyData to an Existing Plot as Points*

Description

Add EventData/PolyData to an existing plot, where each unique EID describes a point.

Usage

```
addPoints (data, xlim = NULL, ylim = NULL, polyProps = NULL,
           cex = NULL, col = NULL, pch = NULL, ...)
```

Arguments

data	EventData or PolyData to add (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
polyProps	PolyData specifying which points to plot and their properties. par parameters passed as direct arguments supersede these data.
cex	vector describing character expansion factors (cycled by EID or PID).
col	vector describing colours (cycled by EID or PID).
pch	vector describing plotting characters (cycled by EID or PID).
...	additional par parameters for the points function.

Details

This function clips data to xlim and ylim before plotting. It only adds PolyData containing X and Y columns.

For additional help on the arguments cex, col, and pch, please see par.

Value

PolyData consisting of the PolyProps used to create the plot.

See Also

combineEvents, convDP, findPolys, locateEvents, plotPoints.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R")) {
  data(nepacLL)
  data(surveyData)
}
#--- plot a map
plotMap(nepacLL, xlim=c(-136, -125), ylim=c(48, 57))
#--- add events
addPoints(surveyData, col=1:7)
```

addPolys *Add a PolySet to an Existing Plot as Polygons*

Description

Add a PolySet to an existing plot, where each unique (PID, SID) describes a polygon.

Usage

```
addPolys (polys, xlim = NULL, ylim = NULL, polyProps = NULL,
          border = NULL, lty = NULL, col = NULL, colHoles = NULL,
          density = NA, angle = NULL, ...)
```

Arguments

polys	PolySet to add (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
polyProps	PolyData specifying which polygons to plot and their properties. par parameters passed as direct arguments supersede these data.
border	vector describing edge colours (cycled by PID).
lty	vector describing line types (cycled by PID).
col	vector describing fill colours (cycled by PID).
colHoles	vector describing hole colours (cycled by PID). The default, NULL, should be used in most cases as it renders holes transparent. colHoles is designed solely to eliminate retrace lines when images are converted to PDF format. If colHoles is specified, underlying information (i.e., previously plotted shapes) will be obliterated. If NA is specified, only outer polygons are drawn, consequently filling holes.
density	vector describing shading line densities (lines per inch, cycled by PID).
angle	vector describing shading line angles (degrees, cycled by PID).
...	additional par parameters for the polygon function.

Details

The plotting routine connects the last vertex of each discrete polygon to the first vertex of that polygon. It supports both borders (border, lty) and fills (col, density, angle). It clips polys to xlim and ylim before plotting.

For additional help on the arguments border, lty, col, density, and angle, please see polygon and par.

Value

PolyData consisting of the PolyProps used to create the plot.

See Also

addLabels, addStipples, clipPolys, closePolys, fixBound, fixPOS, locatePolys, plotLines, plotMap, plotPoints, plotPolys, thinPolys, thickenPolys.

Examples

```
#--- create a PolySet to plot
polys <- data.frame(PID=rep(1,4),POS=1:4,X=c(0,1,1,0),Y=c(0,0,1,1))
polys <- as.PolySet(polys, projection=1)
#--- plot the PolySet
plotPolys(polys,xlim=c(-.5,1.5),ylim=c(-.5,1.5),density=0,projection=1)
#--- add the PolySet to the plot (in a different style)
addPolys(polys, col=3)
```

---

addStipples *Add Stipples to an Existing Plot*

Description

Add stipples to an existing plot.

Usage

```
addStipples (polys, xlim = NULL, ylim = NULL, polyProps = NULL,
             side = 1, density = 1, distance = 4, ...)
```

Arguments

polys	PolySet that provides the stipple boundaries (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
polyProps	PolyData specifying which polygons to stipple and their properties. par parameters passed as direct arguments supersede these data.
side	one of −1, 0, or 1, corresponding to outside, both sides, or inside, respectively.
density	density of points, relative to the default.
distance	distance to offset points, measured as a percentage of the absolute difference in xlim.
...	additional par parameters for the points function.

Details

This function locates stipples based on the PolySet polys and does not stipple degenerate lines.

Value

PolyData consisting of the PolyProps used to create the plot.

See Also

addPoints, addPolys, plotMap, plotPoints, plotPolys, points, PolySet.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R")) {
  data(nepacLL)
}
#--- plot a map
plotMap(nepacLL, xlim=c(-128.66, -122.83), ylim=c(48.00, 51.16))
#--- add stippling
addStipples(nepacLL, col=2, pch=19, cex=0.25)
```

---

appendPolys *Append a Two-Column Matrix to a PolySet*

Description

Append a two-column matrix to a PolySet, assigning PID and possibly SID values automatically or as specified in its arguments.

Usage

```
appendPolys (polys, mat, PID = NULL, SID = NULL, isHole = FALSE)
```

Arguments

polys	existing PolySet; if NULL, creates a new PolySet (*required*).
mat	two-column matrix to append (*required*).
PID	new polygon's PID.
SID	new polygon's SID.
isHole	Boolean value; if TRUE, mat represents a hole.

Details

If the PID argument is NULL, the appended polygon's PID will be one greater than the maximum within polys (if defined); otherwise, it will be 1.

If polys contains an SID column and the SID argument equals NULL, this function uses the next available SID for the corresponding PID.

If polys does not contain an SID column and the caller passes an SID argument, all existing polygons will receive an SID of 1. The new polygon's SID will match the SID argument.

If isHole = TRUE, the polygon's POS values will appropriately represent a hole (reverse order of POS).

If (PID, SID) already exists in the PolySet, the function will issue a warning and duplicate those identifiers.

Value

PolySet containing mat appended to polys. The function retains attributes from polys.

See Also

addPolys, clipPolys, closePolys, convLP, fixBound, fixPOS, joinPolys, plotMap, plotPolys.

Examples

```
#--- create two simple matrices
a <- matrix(data=c(0,0,1,0,1,1,0,1),ncol=2,byrow=TRUE);
b <- matrix(data=c(2,2,3,2,3,3,2,3), ncol=2,byrow=TRUE);
#--- build a PolySet from them
polys <- appendPolys(NULL, a);
polys <- appendPolys(polys, b);
#--- print the result
print (polys);
```

---

bcBathymetry *Data: Bathymetry Spanning British Columbia's Coast*

Description

Bathymetry data spanning British Columbia's coast.

Usage

```
data(bcBathymetry)
```

Format

Three-element list: `x` = vector of horizontal grid line locations, `y` = vector of vertical grid line locations, `z` = (`x` by `y`) matrix containing water depths measured in meters. Positive values indicate distance below sea level and negative values above it.

`contour` and `contourLines` expect data in this format. `convCP` converts the output from `contourLines` into a PolySet.

Note

In R, the data must be loaded using the `data` function.

Source

Bathymetry data acquired from the Scripps Institution of Oceanography at the University of San Diego.

Using their online form, we requested bathymetry data for the complete `nepacLL` region. At forty megabytes, the data were not suitable for distribution in our mapping package. Therefore, we reduced the data to the range $-140° \leq x \leq -122°$ and $47° \leq y \leq 61°$.

References

Smith, W.H.F. and Sandwell, D.T. (1997) Global seafloor topography from satellite altimetry and ship depth soundings. *Science* **277**, 1957–1962.
http://topex.ucsd.edu/WWW_html/mar_topo.html

See Also

`contour`, `contourLines`, `convCP`, `nepacLL`, `nepacLLhigh`.

`calcArea` *Calculate the Areas of Polygons*

Description

Calculate the areas of polygons found in a PolySet.

Usage

```
calcArea (polys, rollup = 3)
```

Arguments

`polys`	PolySet to use.
`rollup`	level of detail in the results; `1` = `PID`s only, by summing all the polygons with the same `PID`, `2` = outer contours only, by subtracting holes from their parent, and `3` = no roll-up.

Details

If `rollup` equals `1`, the results contain an area for each unique `PID` only. When it equals `2`, they contain entries for outer contours only. Finally, setting it to `3` prevents roll-up, and they contain areas for each unique (`PID`, `SID`).

Outer polygons have positive areas and inner polygons negative areas. When polygons are rolled up, the routine sums the positive and negative areas and consequently accounts for holes.

If the PolySet's `projection` attribute equals `"LL"`, the function projects the PolySet in UTM first. If the PolySet's `zone` attribute exists, it uses it for the conversion. Otherwise, it computes the mean longitude and uses that value to determine the zone. The longitude range of zone *i* is $-186 + 6i^\circ < x \leq -180 + 6i^\circ$.

Value

PolyData with columns `PID`, `SID` (*may be missing*), and `area`. If the projection equals `"LL"` or `"UTM"`, the units of area are square kilometres.

See Also

`calcCentroid`, `calcLength`, `calcMidRange`, `calcSummary`, `locatePolys`.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- convert LL to UTM so calculation makes sense
attr(nepacLL, "zone") <- 9
nepacUTM <- convUL(nepacLL)
#--- calculate and print the areas
print(calcArea(nepacUTM))
```

calcCentroid *Calculate the Centroids of Polygons*

Description

Calculate the centroids of polygons found in a PolySet.

Usage

```
calcCentroid (polys, rollup = 3)
```

Arguments

polys	PolySet to use.
rollup	level of detail in the results; 1 = PIDs only, 2 = outer contours only, and 3 = no roll-up. When rollup equals 1 and 2, the function appropriately adjusts for polygons with holes.

Details

If rollup equals 1, the results contain a centroid for each unique PID only. When it equals 2, they contain entries for outer contours only. Finally, setting it to 3 prevents roll-up, and they contain a centroid for each unique (PID, SID).

Value

PolyData with columns PID, SID (*may be missing*), X, and Y.

See Also

calcArea, calcLength, calcMidRange, calcSummary, locateEvents, locatePolys.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- calculate and print the centroids for several polygons
print(calcCentroid(nepacLL[is.element(nepacLL$PID, c(33, 39, 47)), ]))
```

calcConvexHull *Calculate the Convex Hull for a Set of Points*

Description

Calculate the convex hull for a set of points.

Usage

```
calcConvexHull (xydata, keepExtra=FALSE)
```

Arguments

xydata	a data frame with columns X and Y containing spatial coordinates.
keepExtra	logical: if TRUE, retain any additional columns from the input data frame xydata.

Details

This routine uses the function chull() in the package grDevices. By default, it ignores all columns other than X and Y; however, the user can choose to retain additional columns in xydata by specifying keepExtra=TRUE.

Value

PolySet with columns PID, POS, X, Y, and additional columns in xydata if keepExtra=TRUE.

See Also

addPoints, addPolys, calcArea, calcCentroid, calcMidRange, calcSummary, locateEvents, plotMap, plotPoints, plotPolys.

Examples

```
data(surveyData)
#--- plot the convex hull, and then plot the points
plotMap(calcConvexHull(surveyData),col="moccasin")
addPoints(surveyData,col="blue",pch=17,cex=.6);
```

---

calcLength *Calculate the Length of Polylines*

Description

Calculate the length of polylines found in a PolySet.

Usage

```
calcLength (polys, rollup = 3, close = FALSE)
```

Arguments

polys	PolySet to use.
rollup	level of detail in the results; 1 = PIDs only, summing the lengths of each SID within each PID, and 3 = no roll-up. Note: rollup 2 has no meaning in this function and, if specified, will be reset to 3.
close	Boolean value; if TRUE, include the distance between each polygon's last and first vertex, if necessary.

Details

If rollup equals 1, the results contain an entry for each unique PID only. Setting it to 3 prevents roll-up, and they contain an entry for each unique (PID, SID).

If the projection attribute equals "LL", this routine uses Great Circle distances to compute the surface length of each polyline. In doing so, the algorithm simplifies Earth to a sphere.

If the projection attribute equals "UTM" or 1, this routine uses Pythagoras' Theorem to calculate lengths.

Value

PolyData with columns PID, SID (*may be missing*), and length. If projection equals "UTM" or "LL", lengths are in kilometres. Otherwise, lengths are in the same unit as the input PolySet.

See Also

calcArea, calcCentroid, calcMidRange, calcSummary, locatePolys.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- calculate the perimeter of Vancouver Island
print(calcLength(nepacLL[nepacLL$PID == 33, ]))
```

---

calcMidRange *Calculate the Midpoint of the X/Y Ranges of Polygons*

Description

Calculate the midpoint of the X/Y ranges of polygons found in a PolySet.

Usage

```
calcMidRange (polys, rollup = 3)
```

Arguments

polys	PolySet to use.
rollup	level of detail in the results; 1 = PIDs only, 2 = outer contours only, and 3 = no roll-up.

Details

If rollup equals 1, the results contain a mean range for each unique PID only. When it equals 2, they contain entries for outer contours only. Finally, setting it to 3 prevents roll-up, and they contain a mean range for each unique (PID, SID).

Value

PolyData with columns PID, SID (*may be missing*), X, and Y.

See Also

calcArea, calcCentroid, calcLength, calcSummary.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- calculate and print the centroids for several polygons
print(calcMidRange(nepacLL[is.element(nepacLL$PID, c(33, 39, 47)), ]))
```

`calcSummary` *Apply Functions to Polygons in a PolySet*

Description

Apply functions to polygons in a PolySet.

Usage

```
calcSummary (polys, rollup = 3, FUN, ...)
```

Arguments

`polys`	PolySet to use.
`rollup`	level of detail in the results; `1` = `PID`s only, by removing the `SID` column, and then passing each `PID` into `FUN`, `2` = outer contours only, by making hole `SID`s equal to their parent's `SID`, and then passing each (`PID`, `SID`) into `FUN`, and `3` = no roll-up.
`FUN`	the function to apply; it must accept a vector and return a vector or scalar.
`...`	optional arguments for `FUN`.

Details

If `rollup` equals `1`, the results contain an entry for each unique `PID` only. When it equals `2`, they contain entries for outer contours only. Finally, setting it to `3` prevents roll-up, and they contain an entry for each unique (`PID`, `SID`).

Value

PolyData with columns `PID`, `SID` (*may be missing*), `X`, and `Y`. If `FUN` returns a vector of length greater than 1 (say *n*), names the columns `X1`, `X2`, ..., `X`*n* and `Y1`, `Y2`, ..., `Y`*n*.

See Also

`calcArea`, `calcCentroid`, `calcConvexHull`, `calcLength`, `calcMidRange`, `combineEvents`, `findPolys`, `locateEvents`, `locatePolys`, `makeGrid`, `makeProps`.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- calculate and print the centroids for several polygons
print(calcSummary(nepacLL[is.element(nepacLL$PID, c(33, 39, 47)), ],
                          rollup = 3, FUN = mean))
```

`calcVoronoi` *Calculate the Voronoi (Dirichlet) Tesselation for a Set of Points*

Description

Calculate the Voronoi (Dirichlet) tesselation for a set of points.

Usage

```
calcVoronoi (xydata, xlim = NULL, ylim = NULL, eps = 1e-09, frac = 0.0001)
```

Arguments

`xydata`	a data frame with columns `X` and `Y` containing the points.
`xlim`	range of X-coordinates; a bounding box for the coordinates.
`ylim`	range of Y-coordinates; a bounding box for the coordinates.
`eps`	the value of epsilon used in testing whether a quantity is zero.
`frac`	used to detect duplicate input points, which meet the condition $\|x1 - x2\| <$ frac $\times$ (xmax $-$ xmin)and $\|y1 - y2\| <$ frac $\times$ (ymax $-$ ymin).

Details

This routine ignores all columns other than `X` and `Y`.

If the user leaves `xlim` and `ylim` unspecified, the function defaults to the range of the data with each extent expanded by ten percent of the range.

This function sets the attribute `projection` to `1` and the attribute `zone` to `NULL` as it assumes this projection in its calculations.

Value

PolySet with columns `PID`, `POS`, `X`, and `Y`.

See Also

`addPoints`, `addPolys`, `calcArea`, `calcCentroid`, `calcConvexHull`, `calcMidRange`, `calcSummary`, `locateEvents`, `plotMap`, `plotPoints`, `plotPolys`.

Examples

```
#--- create some EventData
events <- as.EventData(data.frame(EID=1:200,
                                  X=rnorm(200),
                                  Y=rnorm(200)),
                       projection=1)
#--- calculate the Voronoi tesselation
polys <- calcVoronoi(events)
#--- create PolyData to color it based on area
polyData <- calcArea(polys)
names(polyData)[is.element(names(polyData), "area")] <- "Z"
colSeq <- seq(0.4, 0.95, length=4)
polyData <- makeProps(polyData,
              breaks=quantile(polyData$Z,c(0,.25,.5,.75,1)),
              propName="col",
```

```
                propVals=rgb(colSeq, colSeq, colSeq))
#--- plot the tesselation
plotMap(polys, polyProps=polyData)
#--- plot the points
addPoints(events, pch=19)
```

`clipLines` *Clip a PolySet as Polylines*

Description

Clip a PolySet, where each unique (`PID`, `SID`) describes a polyline.

Usage

```
clipLines (polys, xlim, ylim, keepExtra = FALSE)
```

Arguments

`polys`	PolySet to clip.
`xlim`	range of X-coordinates.
`ylim`	range of Y-coordinates.
`keepExtra`	Boolean value; if `TRUE`, tries to carry forward any non-standard columns into the result.

Details

For each discrete polyline, the function does not connect vertices 1 and N. It recalculates the `POS` values for each vertex, saving the old values in a column named `oldPOS`. For new vertices, it sets `oldPOS` to `NA`.

Value

PolySet containing the input data, with some points added or removed. A new column `oldPOS` records the original `POS` value for each vertex.

See Also

`clipPolys`, `fixBound`.

Examples

```
#--- create a triangle to clip
polys <- data.frame(PID=rep(1, 3), POS=1:3, X=c(0,1,0), Y=c(0,0.5,1))
#--- clip the triangle in the X direction, and plot the results
plotLines(clipLines(polys, xlim=c(0,.75), ylim=range(polys[, "Y"])))
```

clipPolys *Clip a PolySet as Polygons*

Description

Clip a PolySet, where each unique (PID, SID) describes a polygon.

Usage

```
clipPolys (polys, xlim, ylim, keepExtra = FALSE)
```

Arguments

polys	PolySet to clip.
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
keepExtra	Boolean value; if TRUE, tries to carry forward any non-standard columns into the result.

Details

For each discrete polygon, the function connects vertices 1 and N. It recalculates the POS values for each vertex, saving the old values in a column named oldPOS. For new vertices, it sets oldPOS to NA.

Value

PolySet containing the input data, with some points added or removed. A new column oldPOS records the original POS value for each vertex.

See Also

clipLines, fixBound.

Examples

```
#--- create a triangle that will be clipped
polys <- data.frame(PID=rep(1, 3), POS=1:3, X=c(0,1,.5), Y=c(0,0,1))
#--- clip the triangle in the X direction, and plot the results
plotPolys(clipPolys(polys, xlim=c(0,.75), ylim=range(polys[, "Y"])),
          col=2);
```

closePolys *Close a PolySet*

Description

Close a PolySet of polylines to form polygons.

Usage

```
closePolys (polys)
```

Arguments

polys	PolySet to close.

Details

Generally, run fixBound before this function. The ranges of a PolySet's X and Y columns define the boundary. For each discrete polygon, this function determines if the first and last points lie on a boundary. If both points lie on the same boundary, it adds no points. However, if they lie on different boundaries, it may add one or two corners to the polygon.

When the boundaries are adjacent, one corner will be added as follows:

- top boundary + left boundary implies add top-left corner;
- top boundary + right boundary implies add top-right corner;
- bottom boundary + left boundary implies add bottom-left corner;
- bottom boundary + right boundary implies add bottom-right corner.

When the boundaries are opposite, it first adds the corner closest to a starting or ending polygon vertex. This determines a side (left-right or bottom-top) that connects the opposite boundaries. Then, it adds the other corner of that side to close the polygon.

Value

PolySet identical to polys, except for possible additional corner points.

See Also

fixBound, fixPOS.

Examples

```
#--- 4 corners
polys <- data.frame(PID=c(1, 1, 2, 2, 3, 3, 4, 4),
                    POS=c(1, 2, 1, 2, 1, 2, 1, 2),
                    X = c(0, 1, 2, 3, 0, 1, 2, 3),
                    Y = c(1, 0, 0, 1, 2, 3, 3, 2))
plotPolys(closePolys(polys), col=2)

#--- 2 corners and 1 opposite
polys <- data.frame(PID=c(1, 1, 2, 2, 3, 3, 3),
                    POS=c(1, 2, 1, 2, 1, 2, 3),
                    X = c(0, 1, 0, 1, 5, 6, 1.5),
                    Y = c(1, 0, 2, 3, 0, 1.5, 3))
plotPolys(closePolys(polys), col=2)
```

---

combineEvents *Combine Measurements of Events*

Description

Combine measurements associated with events that occur in the same polygon.

Usage

```
combineEvents (events, locs, FUN, ..., bdryOK = TRUE)
```

Arguments

`events`	EventData with at least four columns (`EID`, `X`, `Y`, `Z`).
`locs`	LocationSet usually resulting from a call to `findPolys`.
`FUN`	a function that produces a scalar from a vector (e.g., `mean`, `sum`).
`...`	optional arguments for `FUN`.
`bdryOK`	Boolean value; if `TRUE`, include boundary points.

Details

This function combines measurements associated with events that occur in the same polygon. Each event (`EID`) has a corresponding measurement `Z`. The `locs` data frame (usually output from `findPolys`) places events within polygons. Thus, each polygon (`PID`, `SID`) determines a set of events within it, and a corresponding vector of measurements `Zv`. The function returns `FUN(Zv)`, a summary of measurements within each polygon.

Value

PolyData with columns `PID`, `SID` (*if in* `locs`), and `Z`.

See Also

`findCells`, `findPolys`, `locateEvents`, `locatePolys`, `makeGrid`, `makeProps`.

Examples

```
#--- create an EventData data frame: let each event have Z = 1
events <- data.frame(EID=1:10, X=1:10, Y=1:10, Z=rep(1, 10))
#--- example output from findPolys where 1 event occurred in the first
#--- polygon, 3 in the second, and 6 in the third
locs <- data.frame(EID=1:10, PID=c(rep(1, 1), rep(2, 3), rep(3, 6)),
                   Bdry=rep(0, 10))
#--- sum the Z column of the events in each polygon, and print the result
print(combineEvents(events=events, locs=locs, FUN=sum))
```

`combinePolys` *Combine Several Polygons into a Single Polygon*

Description

Combine several polygons into a single polygon by modifying the `PID` and `SID` indices.

Usage

```
combinePolys (polys)
```

Arguments

`polys`	PolySet with one or more polygons, each with possibly several components/holes.

Details

This function accepts a PolySet containing one or more polygons (`PID`s), each with one or more components or holes (`SID`s). The `SID` column need not exist in the input. The function combines these polygons into a single polygon by simply renumbering the `PID` and `SID` indices. The resulting PolySet contains a single `PID` (with the value 1) and uses the `SID` value to differentiate between polygons, their components, and holes.

Value

PolySet, possibly with the addition of an SID column if it did not already exist. The function may also reorder columns such that PID, SID, POS, X and Y appear first, in that order.

See Also

dividePolys

---

convCP *Convert Contour Lines into a PolySet*

Description

Convert output from contourLines into a PolySet.

Usage

```
convCP (data, projection = NULL, zone = NULL)
```

Arguments

data	contour line data, often from the contourLines function.
projection	optional projection attribute to add to the PolySet.
zone	optional zone attribute to add to the PolySet.

Details

data contains a list as described below. The contourLines function create a list suitable for the data argument.

A three-element list describes each contour. The named elements in this list include the scalar level, the vector x, and the vector y. Vectors x and y must have equal lengths. A higher-level list (data) contains one or more of these contours lists.

Value

A list with two named elements PolySet and PolyData. The PolySet element contains a PolySet representation of the contour lines. The PolyData element links each contour line (PID, SID) with a level.

See Also

contour, contourLines, convLP, makeTopography.

Examples

```
#--- create sample data for the contourLines() function
x <- seq(-0.5, 0.8, length=50);
y <- x;
z <- outer(x, y, FUN = function(x,y) { sin(2*pi*(x^2+y^2))^2; } );
data <- contourLines(x, y, z, levels=c(0.2, 0.8));
#--- pass that sample data into convCP()
result <- convCP(data);
#--- plot the result
plotLines(result$PolySet, projection=1);
print(result$PolyData);
```

convDP *Convert EventData/PolyData into a PolySet*

Description

Convert EventData/PolyData into a PolySet.

Usage

```
convDP (data, xColumns, yColumns)
```

Arguments

data	PolyData or EventData.
xColumns	vector of X-column names.
yColumns	vector of Y-column names.

Details

This function expects data to contain several X- and Y-columns. For example, consider data with columns x1, y1, x2, and y2. Suppose xColumns = c("x1", "x2") and yColumns = c("y1", "y2"). The result will contain nrow(data) polygons. Each one will have two vertices, (x1, y1) and (x2, y2) and POS values 1 and 2, respectively. If data includes an SID column, so will the result.

If data contains an EID and not a PID column, the function uses the EIDs as PIDs.

If data contains both PID and EID columns, the function assumes it is PolyData and ignores the EID column.

Value

PolySet with the same PIDs as those given in data. If data has an SID column, the result will include it.

See Also

addPoints, plotPoints.

Examples

```
#--- create sample PolyData
polyData <- data.frame(PID=c(1, 2, 3),
                       x1=c(1, 3, 5), y1=c(1, 3, 2),
                       x2=c(1, 4, 5), y2=c(2, 4, 1),
                       x3=c(2, 4, 6), y3=c(2, 3, 1));

#--- print PolyData
print(polyData);
#--- make a PolySet from PolyData
polys <- convDP(polyData,
                xColumns=c("x1", "x2", "x3"),
                yColumns=c("y1", "y2", "y3"));
#--- print and plot the PolySet
print(polys);
plotLines(polys, xlim=c(0, 7), ylim=c(0, 5), col=2);
```

convLP *Convert Polylines into a Polygon*

Description

Convert two polylines into a polygon.

Usage

```
convLP (polyA, polyB, reverse = TRUE)
```

Arguments

polyA	PolySet containing a polyline.
polyB	PolySet containing a polyline.
reverse	Boolean value; if TRUE, reverse polyB's vertices.

Details

The resulting PolySet contains all the vertices from polyA in their original order. If reverse = TRUE, this function appends the vertices from polyB in the reverse order (nrow(polyB):1). Otherwise, it appends them in their original order. The PID column equals the PID of polyA. No SID column appears in the result. The resulting polygon is an exterior boundary.

Value

PolySet with a single PID that is the same as polyA. The result contains all the vertices in polyA and polyB. It has the same projection and zone attributes as those in the input PolySets. If an input PolySet's attributes equal NULL, the function uses the other PolySet's. If the PolySet attributes conflict, the result's attribute equals NULL.

See Also

addLines, appendPolys, closePolys, convCP, joinPolys, plotLines.

Examples

```
#--- create two polylines
polyline1 <- data.frame(PID=rep(1, 2), POS=1:2, X=c(1, 4), Y=c(1, 4));
polyline2 <- data.frame(PID=rep(1, 2), POS=1:2, X=c(2, 5), Y=c(1, 4));

#--- create two plots to demonstrate the effect of `reverse'
par(mfrow=c(2, 1))
plotPolys(convLP(polyline1, polyline2, reverse = TRUE), col=2);
plotPolys(convLP(polyline1, polyline2, reverse = FALSE), col=3);
```

convUL *Convert Coordinates between UTM and Lon/Lat*

Description

Convert coordinates between UTM and Lon/Lat.

Usage

```
convUL (xydata, km=TRUE, southern=NULL)
```

Arguments

xydata	data frame with columns X and Y.
km	Boolean value; if TRUE, UTM coordinates within xydata are in kilometres; otherwise, metres.
southern	Boolean value; if TRUE, forces conversions from UTM to longitude/latitude to produce coordinates within the southern hemisphere. For conversions from UTM, this argument defaults to FALSE. For conversions from LL, the function determines southern from xydata.

Details

The object xydata must possess a projection attribute that identifies the current projection. If the data frame contains UTM coordinates, it must also have a zone attribute equal to a number between 1 and 60 (inclusive). If it contains geographic (longitude/latitude) coordinates and the zone attribute is missing, the function computes the mean longitude and uses that value to determine the zone. The longitude range of zone *i* is $-186 + 6i^\circ < x \leq -180 + 6i^\circ$.

This function converts the X and Y columns of xydata from "LL" to "UTM" or vice-versa. If the data span more than **one** zone to the right or left of the intended central zone, the underlying algorithm may produce erroneous results. This limitation means that the user should use the most central zone of the mapped region, or allow the function to determine the central zone when converting from geographic to UTM coordinates. After the conversion, this routine adjusts the data frame's attributes accordingly.

Value

A data frame identical to xydata, except that the X and Y columns contain the results of the conversion, and the projection attribute matches the new projection.

Author(s)

Nicholas Boers, Dept. of Computer Science, Grant MacEwan University, Edmonton AB

References

Ordnance Survey. (2010) A guide to coordinate systems in Great Britain. *Report D00659* **(v2.1)**. Southampton, UK.
http://www.ordnancesurvey.co.uk/oswebsite/gps/docs/A_Guide_to_Coordinate_Systems_in_Gre

See Also

closePolys, fixBound.

Examples

```
#--- load the data
data(nepacLL, package="PBSmapping")
#--- set the zone attribute
#--- use a zone that is most central to the mapped region
attr(nepacLL, "zone") <- 6
#--- convert and plot the result
nepacUTM <- convUL(nepacLL)
plotMap(nepacUTM)
```

---

dividePolys *Divide a Single Polygon into Several Polygons*

Description

Divide a single polygon (with several outer-contour components) into several polygons, a polygon for each outer contour, by modifying the PID and SID indices.

Usage

```
dividePolys (polys)
```

Arguments

polys	PolySet with one or more polygons, each with possibly several components/holes.

Details

Given the input PolySet, this function renumbers the PID and SID indices so that each outer contour has a unique PID and is followed by all of its holes, identifying them with SIDs greater than one.

Value

PolySet, possibly with the addition of an SID column if it did not already exist. The function may also reorder columns such that PID, SID, POS, X and Y appear first, in that order.

See Also

combinePolys.

EventData *EventData Objects*

Description

PBS Mapping functions that expect EventData will accept properly formatted data frames in their place (see 'Details').

as.EventData attempts to coerce a data frame to an object with class EventData.

is.EventData returns TRUE if its argument is of class EventData.

Usage

```
as.EventData(x, projection = NULL, zone = NULL)
is.EventData(x, fullValidation = TRUE)
```

Arguments

`x`	data frame to be coerced or tested.
`projection`	optional `projection` attribute to add to EventData, possibly overwriting an existing attribute.
`zone`	optional `zone` attribute to add to EventData, possibly overwriting an existing attribute.
`fullValidation`	Boolean value; if `TRUE`, fully test `x`.

Details

We define EventData as a data frame with at least three fields named (`EID`, `X`, `Y`). Conceptually, an EventData object describes events that take place at specific points (`X`, `Y`) in two-dimensional space. Additional fields specify measurements associated with these events. For example, in a fishery context EventData could describe fishing events associated with trawl tows, based on the fields:

- `EID` - fishing event (tow) identification number;
- `X`, `Y` - fishing location;
- `Duration` - length of time for the tow;
- `Depth` - average depth of the tow;
- `Catch` - biomass captured.

Like PolyData, EventData can have attributes `projection` and `zone`, which may be absent. Inserting the string `"EventData"` as the class attribute's first element alters the behaviour of some functions, including `print` (if `PBSprint` is `TRUE`) and `summary`.

Value

The `as.EventData` method returns an object with classes `"EventData"` and `"data.frame"`, in that order.

See Also

LocationSet, PolyData, PolySet.

`extractPolyData` *Extract PolyData from a PolySet*

Description

Extract PolyData from a PolySet. Columns for the PolyData include those other than `PID`, `SID`, `POS`, `oldPOS`, `X`, and `Y`.

Usage

```
extractPolyData (polys)
```

Arguments

polys	PolySet to use.

Details

This function identifies the PolySet's extra columns and determines if those columns contain unique values for each (PID, SID). Where they do, the (PID, SID) will appear in the PolyData output with that unique value. Where they do not, the extra column will contain NAs for that (PID, SID).

Value

PolyData with columns PID, SID, and any extra columns.

See Also

makeProps, PolyData, PolySet.

Examples

```
#--- create a PolySet with an extra column
polys <- data.frame(PID = c(rep(1, 10), rep(2, 10)),
                    POS = c(1:10, 1:10),
                    X = c(rep(1, 10), rep(1, 10)),
                    Y = c(rep(1, 10), rep(1, 10)),
                    colour = (c(rep("green", 10), rep("red", 10))));
#--- extract the PolyData
print(extractPolyData(polys))
```

---

findCells — *Find the Grid Cells that Contain Events*

Description

Find the grid cells in a PolySet that contain events specified in EventData. Similar to findPolys, except this function requires a PolySet resulting from makeGrid. This restriction allows this function to calculate the result with greater efficiency.

Usage

```
findCells (events, polys)
```

Arguments

events	EventData to use.
polys	PolySet to use.

Details

The resulting data frame, a LocationSet, contains the columns EID, PID, SID (*if in* polys), and Bdry, where an event (EID) occurs in a polygon (PID, SID). The Boolean variable Bdry indicates whether an event lies on a polygon's edge. Note that if an event lies properly outside of all the polygons, then a record with (EID, PID, SID) does not occur in the output. It may happen, however, that an event occurs in multiple polygons (i.e., on two or more boundaries). Thus, the same EID can occur more than once in the output.

Value

LocationSet that links events with polygons.

See Also

combineEvents, findPolys, locateEvents, locatePolys, LocationSet, makeGrid.

Examples

```
#--- create some EventData: points in a diagonal line
events <- data.frame(EID=1:11, X=seq(0, 2, length=11),
                     Y=seq(0, 2, length=11))
events <- as.EventData(events, projection=1);
#--- create a PolySet (a grid)
polys <- makeGrid (x=seq(0, 2, by=0.50), y=seq(0, 2, by=0.50),
                   projection=1)
#--- show a picture
plotPolys(polys, xlim=range(polys$X)+c(-0.1, 0.1),
          ylim=range(polys$Y)+c(-0.1, 0.1), projection=1)
addPoints(events, col=2);
#--- run findCells and print the results
fc <- findCells(events, polys)
fc <- fc[order(fc$EID, fc$PID, fc$SID), ]
fc$label <- paste(fc$PID, fc$SID, sep=", ")
print (fc)
#--- add labels to the graph
addLabels(as.PolyData(fc[!duplicated(paste(fc$PID,fc$SID)), ],
                      projection=1),
          placement="CENTROID", polys=as.PolySet(polys, projection=1),
          col=4)
```

---

findPolys *Find the Polygons that Contain Events*

Description

Find the polygons in a PolySet that contain events specified in EventData.

Usage

```
findPolys (events, polys, maxRows = 1e+05)
```

Arguments

events	EventData to use.
polys	PolySet to use.
maxRows	estimated maximum number of rows in the output LocationSet.

Details

The resulting data frame, a LocationSet, contains the columns EID, PID, SID (*if in* polys), and Bdry, where an event (EID) occurs in a polygon (PID, SID) and SID does not correspond to an inner boundary. The Boolean variable Bdry indicates whether an event lies on a polygon's edge. Note that if an event lies properly outside of all the polygons, then a record with (EID, PID, SID) does not occur in the output. It may happen, however, that an event occurs in multiple polygons. Thus, the same EID can occur more than once in the output.

Value

LocationSet that links events with polygons.

See Also

`combineEvents`, `findCells`, `locateEvents`, `locatePolys`, LocationSet, `makeGrid`.

Examples

```
#--- create some EventData: a column of points at X = 0.5
events <- data.frame(EID=1:10, X=.5, Y=seq(0, 2, length=10))
events <- as.EventData(events, projection=1);
#--- create a PolySet: two squares with the second above the first
polys <- data.frame(PID=c(rep(1, 4), rep(2, 4)), POS=c(1:4, 1:4),
                    X=c(0, 1, 1, 0, 0, 1, 1, 0),
                    Y=c(0, 0, 1, 1, 1, 1, 2, 2))
polys <- as.PolySet(polys, projection=1);
#--- show a picture
plotPolys(polys, xlim=range(polys$X)+c(-0.1, 0.1),
          ylim=range(polys$Y)+c(-0.1, 0.1), projection=1);
addPoints(events, col=2);
#--- run findPolys and print the results
print(findPolys(events, polys))
```

---

fixBound *Fix the Boundary Points of a PolySet*

Description

The ranges of a PolySet's `X` and `Y` columns define its boundary. This function fixes a PolySet's vertices by moving vertices near a boundary to the actual boundary.

Usage

```
fixBound (polys, tol)
```

Arguments

`polys`	PolySet to fix.
`tol`	vector (length 1 or 2) specifying a percentage of the ranges to use in defining *near* to a boundary. If `tol` has two elements, the first specifies the tolerance for the x-axis and the second the y-axis. If it has only one element, the function uses the same tolerance for both axes.

Details

When moving vertices to a boundary, the function moves them strictly horizontally or vertically, as appropriate.

Value

PolySet identical to the input, except for possible changes in the `X` and `Y` columns.

See Also

`closePolys`, `fixPOS`, `isConvex`, `isIntersecting`, PolySet.

Examples

```
#--- set up a long horizontal and long vertical line to extend the plot's
#--- limits, and then try fixing the bounds of a line in the top-left
#--- corner and a line in the bottom-right corner
polys <- data.frame(PID=c(1, 1, 2, 2,  3, 3, 4, 4),
                    POS=c(1, 2, 1, 2,  1, 2, 1, 2),
                    X = c(0, 10, 5, 5, 0.1, 4.9, 5.1, 9.9),
                    Y = c(5, 5, 0, 10, 5.1, 9.9, 0.1, 4.9))
polys <- fixBound(polys, tol=0.0100001)
plotLines(polys)
```

---

fixPOS *Fix the POS Column of a PolySet*

Description

Fix the POS column of a PolySet by recalculating it using sequential integers.

Usage

```
fixPOS (polys, exteriorCCW = NA)
```

Arguments

polys	PolySet to fix.
exteriorCCW	Boolean value; if TRUE, orders exterior polygon vertices in a counter-clockwise direction. If FALSE, orders them in a clockwise direction. If NA, maintains their original order.

Details

This function recalculates the POS values of each (PID, SID) as either 1 to N or N to 1, depending on the order of POS (ascending or descending) in the input data. POS values in the input must be properly ordered (ascending or descending), but they may contain fractional values. For example, POS = 2.5 might correspond to a point manually added between POS = 2 and POS = 3. If exteriorCCW = NA, all other columns remain unchanged. Otherwise, it orders the X and Y columns according to exteriorCCW.

Value

PolySet with the same columns as the input, except for possible changes to the POS, X, and Y columns.

See Also

closePolys, fixBound, isConvex, isIntersecting, PolySet.

Examples

```
#--- create a PolySet with broken POS numbering
polys <- data.frame(PID = c(rep(1, 10), rep(2, 10)),
                    POS = c(seq(2, 10, length = 10), seq(10, 2, length = 10)),
                    X = c(rep(1, 10), rep(1, 10)),
                    Y = c(rep(1, 10), rep(1, 10)))
#--- fix the POS numbering
polys <- fixPOS(polys)
#--- print the results
print(polys)
```

importEvents *Import EventData from a Text File*

Description

Import a text file and convert into EventData.

Usage

```
importEvents(EventData, projection=NULL, zone=NULL)
```

Arguments

EventData	filename of EventData text file.
projection	optional projection attribute to add to EventData.
zone	optional zone attribute to add to EventData.

Value

An imported EventData.

See Also

importPolys, importLocs, importGSHHS, importShapefile

importGSHHS *Import Data from a GSHHS Database*

Description

Import data from a GSHHS database and convert data into a PolySet with a PolyData attribute.

Usage

```
importGSHHS(gshhsDB, xlim, ylim, maxLevel=4, n=0)
```

Arguments

gshhsDB	path name to binary GSHHS database. If unspecified, looks for gshhs_f.b in the root of the PBSmapping library directory.
xlim	range of X-coordinates (for clipping). The range should be between 0 and 360.
ylim	range of Y-coordinates (for clipping).
maxLevel	maximum level of polygons to import: 1 (land), 2 (lakes on land), 3 (islands in lakes), or 4 (ponds on islands); ignored when importing lines.
n	minimum number of vertices that must exist in a line/polygon in order for it to be imported.

Details

This routine requires a binary GSHHS (Global Self-consistent, Hierarchical, High-resolution Shoreline) database file. The GSHHS database has been released in the public domain and may be downloaded from http://www.soest.hawaii.
At the time of writing, the most recent database is gshhs+wdbii_2.2.0.zip.

The database gshhs+wdbii_2.2.0.zip contains geographical coordinates for shorelines (gshhs), rivers (wbd_rivers), and borders (wdb_borders). The latter two come from World DataBank II (WDBII):
http://meta.wikimedia.org/wiki/Geographical_data#CIA_World_DataBank_II_and_derivates
The five resolutions available are:
full (f), high (h), intermediate (i), low (l), and coarse (c).

This routine returns a PolySet object with an associated PolyData attribute. The attribute contains four fields: (a) PID, (b) SID, (c) Level, and (d) Source. Each record corresponds to a line/polygon in the PolySet. The Level indicates the line's/polygon's level (1=land, 2=lake, 3=island, 4=pond). The Source identifies the data source (1=WVS, 0=CIA (WDBII)).

Value

A PolySet with a PolyData attribute.

Author(s)

Nicholas Boers, Dept. of Computer Science, Grant MacEwan University, Edmonton AB

See Also

importEvents, importLocs, importPolys, importShapefile

Examples

```
## Not run:
pbsfun = function(ex=1) {
  switch(ex, {
  #--- EXAMPLE 1
  #--- set some limits appropriate for a map of Canada
  limits <- list(x = c(216.0486, 307.1274), y = c(42.87209, 77.35183))
  #--- extract data from the GSHHS binary files; you will need to download
  #--- these files from http://www.soest.hawaii.edu/wessel/gshhs/
  #--- and place them in an appropriate location
  polys <- importGSHHS ("./gshhs+wdbii_2.2.0/gshhs/gshhs_l.b",
                  xlim=limits$x, limits$y, maxLevel=4)
  rivers <- importGSHHS ("./gshhs+wdbii_2.2.0/gshhs/wdb_rivers_i.b",
                         xlim=limits$x, limits$y)
  borders <- importGSHHS ("./gshhs+wdbii_2.2.0/gshhs/wdb_borders_i.b",
                          xlim=limits$x, limits$y)
  #--- create a PNG for the output
  png ("./Canada.png", width=1600, height=1200, pointsize=24)
  #--- plot the polygons, river, and then borders
  plotMap (polys, plt=c(.05,.99,.075,.99), col="moccasin", bg="skyblue")
  addLines (rivers, col="lightblue")
  addLines (borders, col="red")
  #--- close the output file
  dev.off ()
  },{
  #--- EXAMPLE 2
  #--- clip out Manitoulin Island area which includes all four levels
  polys <- importGSHHS ("./gshhs+wdbii_2.2.0/gshhs/gshhs_f.b",
                        xlim=c(276, 279), ylim=c(45.3, 46.5), maxLevel=4)
```

```
  #--- plot the map and add a label
  plotMap (polys, col="beige", bg="lightblue");
  text (-82.08, 45.706, "Manitoulin Isl")
  })
  invisible()
}
pbsfun(1); pbsfun(2)

## End(Not run)
```

`importLocs` *Import LocationSet from a text file*

Description

Import a text file and convert into a LocationSet.

Usage

```
importLocs(LocationSet)
```

Arguments

`LocationSet`	filename of LocationSet text file.

Value

An imported LocationSet.

See Also

`importPolys`, `importEvents`, `importGSHHS`, `importShapefile`

`importPolys` *Import PolySet from a text file*

Description

Import a text file and convert into a PolySet with optional PolyData attribute.

Usage

```
importPolys(PolySet, PolyData=NULL, projection=NULL, zone=NULL)
```

Arguments

`PolySet`	filename of PolySet text file.
`PolyData`	optional filename of PolyData text file.
`projection`	optional `projection` attribute to add to EventData.
`zone`	optional `zone` attribute to add to EventData.

Value

An imported PolySet with optional PolyData attribute.

See Also

importEvents, importLocs, importGSHHS, importShapefile

---

importShapefile *Import an ESRI Shapefile*

Description

Import an ESRI shapefile (.shp) into either a PolySet or EventData.

Usage

```
importShapefile (fn, readDBF=TRUE, projection=NULL, zone=NULL,
     placeholes=FALSE, minverts=3)
```

Arguments

fn	file name of the shapefile to import; specifying the extension is optional.
readDBF	Boolean value; if TRUE, it also imports the .dbf (a database containing the feature attributes) associated with the shapefile.
projection	optional projection attribute to override the internally derived value.
zone	optional zone attribute to override the default value of NULL.
placeholes	logical: if TRUE then for every PID identify solids and holes, and place holes under appropriate solids.
minverts	minimum number of vertices required for a polygon representing a hole to be retained (does not affect solids).

Details

This routine imports an ESRI shapefile (.shp) into either a PolySet or EventData, depending on the type of shapefile. It supports types 1 (Point), 3 (PolyLine), and 5 (Polygon) and imports type 1 into EventData and types 3 and 5 into a PolySet. In addition to the shapefile (.shp), it requires the related index file (.shx).

If a database containing feature attributes (.dbf) exists, it also imports this database by default. For EventData, it binds the database columns to the EventData object. For a PolySet, it saves the database in a PolyData object and attaches that object to the PolySet in an attribute named "PolyData".

If a .prj file exists, this information is attached as an attribute. If the first 3 characters are 'GEO', then a geographic projection is assumed and projection="LL". If the first 4 characters are 'PROJ', and 'UTM' occurs elsewhere in the string, then the Universal Transverse Mercator projection is assumed and projection="UTM". Otherwise, projection=1.

If an .xml file exists, this information is attached as an attribute.

Shapes of numeric shape type 5 exported from **ArcView** in geographic projection identify solids as polygons with vertices following a clockwise path and holes as polygons that follow a counter-clockwise path. Unfortuantely, either the export from **ArcView** or the import using a C-routine from the package **maptools** often does not report solids followed by their holes. We employ a new R function placeHoles to do this for us. Ideally, this routine should be rendered in C, but for now we use this function if the user sets the argument placeholes=TRUE. Depending on the size and complexity of your shapefile, the computation may take a while.

Value

For points, EventData with columns EID, X, and Y, possibly with other columns from the attribute database. For polylines and polygons, a PolySet with columns PID, SID, POS, X, Y and attribute projection. Other attributes that may or may not be attached: parent.child (boolean vector from original input), shpType (numeric shape type: 1, 3, or 5), prj (projection information from .prj file, xml (metadata from an .xml file), PolyData (data from the attribute database .dbf), and zone (UTM zone).

See Also

importGSHHS, importEvents, importLocs, importPolys, placeHoles
In the package **sp**, see the function point.in.polygon

---

isConvex *Determine Whether Polygons are Convex*

Description

Determine whether polygons found in a PolySet are convex.

Usage

```
isConvex (polys)
```

Arguments

polys	PolySet to use.

Details

Convex polygons do not self-intersect. In a convex polygon, only the first and last vertices may share the same coordinates (i.e., the polygons are optionally closed).

The function does not give special consideration to holes. It returns a value for each unique (PID, SID), regardless of whether a contour represents a hole.

Value

PolyData with columns PID, SID (*may be missing*), and convex. Column convex contains Boolean values.

See Also

isIntersecting, PolySet.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL);
#--- calculate then print the polygons that are convex
p <- isConvex(nepacLL);
#--- nepacLL actually contains no convex polygons
print(p[p$convex, ]);
```

`isIntersecting` *Determine Whether Polygons are Self-Intersecting*

Description

Determine whether polygons found in a PolySet are self-intersecting.

Usage

```
isIntersecting (polys, numericResult = FALSE)
```

Arguments

`polys`	PolySet to use.
`numericResult`	Boolean value; if `TRUE`, returns the number of intersections.

Details

When `numericResult = TRUE`, this function counts intersections as the algorithm processes them. It counts certain types (i.e., those involving vertices and those where an edge retraces over an edge) more than once.

The function does not give special consideration to holes. It returns a value for each unique (`PID`, `SID`), regardless of whether a contour represents a hole.

Value

PolyData with columns `PID`, `SID` (*may be missing*), and `intersecting`. If `numericResult` is `TRUE`, `intersecting` contains the number of intersections. Otherwise, it contains a Boolean value.

See Also

`isConvex`, PolySet.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL);
#--- calculate then print the polygons that are self-intersecting
p <- isIntersecting(nepacLL, numericResult = FALSE);
print(p[p$intersecting, ]);
```

joinPolys *Join One or Two PolySets using a Logic Operation*

Description

Join one or two PolySets using a logic operation.

Usage

```
joinPolys(polysA,polysB=NULL,operation="INT",maxVert=1e+05)
```

Arguments

polysA	PolySet to join.
polysB	optional second PolySet with which to join.
operation	one of "DIFF", "INT", "UNION", or "XOR", representing difference, intersection, union, and exclusive-or, respectively.
maxVert	estimated maximum number of vertices in the output PolySet.

Details

This function interfaces with the General Polygon Clipper library (http://www.cs.man.ac.uk/aig/staff/alan/softw produced by Alan Murta at the University of Manchester. Consequently, we adopt some of his terminology in the details below.

Murta (2004) defines a *generic polygon* (or *polygon set*) as zero or more disjoint boundaries of arbitrary configuration. He relates a *boundary* to a contour, where each may be convex, concave or self-intersecting. In a PolySet, the polygons associated with each unique PID loosely correspond to a generic polygon, as they can represent both inner and outer boundaries. Our use of the term *generic polygon* includes the restrictions imposed by a PolySet. For example, the polygons for a given PID cannot be arranged arbitrarily.

If polysB is NULL, this function sequentially applies the operation between the generic polygons in polysA. For example, suppose polysA contains three generic polygons (A, B, C). The function outputs the PolySet containing ((A op B) op C).

If polysB is not NULL, this function applies operation between each generic polygon in polysA and each one in polysB. For example, suppose polysA contains two generic polygons (A, B) and polysB contains two generic polygons (C, D). The function's output is the concatenation of A *op* C, B *op* C, A *op* D, B *op* D, with PIDs 1 to 4, respectively. Generally there are n times m comparisons, where n = number of polygons in polysA and m = number of polygons in polysB. If polysB contains only one generic polygon, the function maintains the PIDs from polysA. It also maintains them when polysA contains only one generic polygon and the operation is difference. Otherwise, if polysA contains only one generic polygon, it maintains the PIDs from polysB.

Value

If polysB is NULL, the resulting PolySet contains a single generic polygon (one PID), possibly with several components (SIDs). The function recalculates the PID and SID columns.

If polysB is not NULL, the resulting PolySet contains one or more generic polygons (PIDs), each with possibly several components (SIDs). The function recalculates the SID column, and depending on the input, it may recalculate the PID column.

References

Murta, A. (2004) *A General Polygon Clipping Library*. Accessed: July 29, 2004.
http://www.cs.man.ac.uk/aig/staff/alan/software/gpc.html

See Also

addPolys, appendPolys, clipPolys, closePolys, fixBound, fixPOS, locatePolys, plotMap, plotPoints, thickenPolys, thinPolys.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- create a triangle to use in clipping
polysB <- data.frame(PID=rep(1, 3), POS=1:3,
  X=c(-127.5, -124.5, -125.6), Y = c(49.2, 50.3, 48.6))
#--- intersect nepacLL with the single polygon, and plot the result
plotMap(joinPolys(nepacLL, polysB), col=5)
#--- add nepacLL in a different line type to emphasize the intersection
addPolys(nepacLL, border=2, lty=8, density=0)
```

---

locateEvents *Locate Events on the Current Plot*

Description

Locate events on the current plot (using the locator function).

Usage

```
locateEvents (EID, n = 512, type = "p", ...)
```

Arguments

EID	vector of event IDs (*optional*).
n	maximum number of events to locate.
type	one of "n", "p", "l", or "o". If "p" or "o", then the points are plotted; if "l" or "o", then the points are joined by lines.
...	additional par parameters for the locator function.

Details

This function allows its user to define events with mouse clicks on the current plot via the locator function. The arguments n and type are the usual parameters of the locator function. If EID is not missing, then n = length(EID).

On exit from locator, suppose the user defined *m* events. If EID was missing, then the output data frame will contain *m* events. However, if EID exists, then the output data frame will contain length(EID) events, and both X and Y will be NA for events EID[(*m*+1):n]. The na.omit function can remove rows with NAs.

Value

EventData with columns EID, X, and Y, and projection attribute equal to the map's projection. The function does not set the zone attribute.

See Also

addPoints, combineEvents, convDP, EventData, findCells, findPolys, plotPoints.

Examples

```
#--- define five events on the current plot, numbering them 10 to 14
## Not run: events <- locateEvents(EID = 10:14)
```

---

locatePolys *Locate Polygons on the Current Plot*

Description

Locate polygons on the current plot (using the locator function).

Usage

```
locatePolys (pdata, n = 512, type = "o", ...)
```

Arguments

pdata	PolyData (*optional*) with columns PID and SID (*optional*), with two more optional columns n and type.
n	maximum number of points to locate.
type	one of "n", "p", "l", or "o". If "p" or "o", then the points are plotted; if "l" or "o", then the points are joined by lines.
...	additional par parameters for the locator function.

Details

This function allows its user to define polygons with mouse clicks on the current plot via the locator function. The arguments n and type are the usual parameters for the locator function, but the user can specify them for each individual (PID, SID) in a pdata object.

If a pdata object exists, the function ignores columns other than PID, SID, n, and type. If pdata includes n, then an outer boundary has n > 0 and an inner boundary has n < 0.

On exit from locator, suppose the user defined m vertices for a given polygon. For that polygon, the X and Y columns will contain NAs where POS = (m+1):n for outer-boundaries and POS = (|n|−m):1 for inner-boundaries. The na.omit function can remove rows with NAs.

If a pdata object does not exist, the output contains only one polygon with a PID equal to 1. One inner-boundary polygon (POS goes from n to 1) can be generated by supplying a negative n.

If type = "o" or type = "l", the function draws a line connecting the last and first vertices.

Value

PolySet with projection attribute equal to the map's projection. The function does not set the zone attribute.

See Also

addPolys, appendPolys, clipPolys, closePolys, findCells, findPolys, fixPOS, joinPolys, plotMap, plotPolys, thickenPolys, thinPolys.

Examples

```
#--- define one polygon with up to 5 vertices on the current plot
## Not run: polys <- locatePolys(n = 5)
```

LocationSet *LocationSet Objects*

Description

PBS Mapping functions that expect LocationSet's will accept properly formatted data frames in their place (see 'Details').

`as.LocationSet` attempts to coerce a data frame to an object with class LocationSet.

`is.LocationSet` returns `TRUE` if its argument is of class LocationSet.

Usage

```
as.LocationSet(x)
is.LocationSet(x, fullValidation = TRUE)
```

Arguments

`x`	data frame to be coerced or tested.
`fullValidation`	Boolean value; if `TRUE`, fully test `x`.

Details

A PolySet can define regional boundaries for drawing a map, and EventData can give event points on the map. Which events occur in which regions? Our function `findPolys` resolves this problem. The output lies in a LocationSet, a data frame with three or four columns (`EID`, `PID`, `SID`, `Bdry`), where `SID` may be missing. One row in a LocationSet means that the event `EID` occurs in the polygon (`PID`, `SID`). The boundary (`Bdry`) field specifies whether (`Bdry=T`) or not (`Bdry=F`) the event lies on the polygon boundary. If `SID` refers to an inner polygon boundary, then `EID` occurs in (`PID`, `SID`) only if `Bdry=T`. An event may occur in multiple polygons. Thus, the same `EID` can occur in multiple records. If an `EID` does not fall in any (`PID`, `SID`), or if it falls within a hole, it does not occur in the output LocationSet. Inserting the string `"LocationSet"` as the first element of a LocationSet's `class` attribute alters the behaviour of some functions, including `print` (if `PBSprint` is `TRUE`) and `summary`.

Value

The `as.LocationSet` method returns an object with classes `"LocationSet"` and `"data.frame"`, in that order.

See Also

EventData, PolyData, PolySet.

makeGrid *Make a Grid of Polygons*

Description

Make a grid of polygons, using PIDs and SIDs according to the input arguments.

Usage

```
makeGrid(x,y,byrow=TRUE,addSID=TRUE,projection=NULL,zone=NULL)
```

Arguments

x	vector of X-coordinates (of length m).
y	vector of Y-coordinates (of length n).
byrow	Boolean value; if TRUE, increment PID along X.
addSID	Boolean value; if TRUE, include an SID column in the resulting PolySet.
projection	optional projection attribute to add to the PolySet.
zone	optional zone attribute to add to the PolySet.

Details

This function makes a grid of polygons, labeling them according to byrow and addSID. In the following description, the variables i and j indicate column and row numbers, respectively, where the lower-left cell of the grid is (1, 1).

- byrow = TRUE and addSID = FALSE implies PID $= i + (j-1) \times (m-1)$
- byrow = FALSE and addSID = FALSE implies PID $= j + (i-1) \times (n-1)$
- byrow = TRUE and addSID = TRUE implies PID $= i$, SID $= j$
- byrow = FALSE and addSID = TRUE implies PID $= j$, SID $= i$

Value

PolySet with columns PID, SID (*if* addSID = TRUE), POS, X, and Y. The PolySet is a set of rectangular grid cells with vertices:
$(x_i, y_j), (x_{i+1}, y_j), (x_{i+1}, y_{j+1}), (x_i, y_{j+1})$.

See Also

addPolys, clipPolys, combineEvents, findCells, findPolys, PolySet, thickenPolys.

Examples

```
#--- make a 10 x 10 grid
polyGrid <- makeGrid(x=0:10, y=0:10)
#--- plot the grid
plotPolys(polyGrid, density=0, projection=1)
```

makeProps *Make Polygon Properties*

Description

Append a column for a polygon property (e.g., `border` or `lty`) to PolyData based on measurements in the PolyData's `Z` column.

Usage

```
makeProps(pdata,breaks,propName="col",propVals=1:(length(breaks)-1))
```

Arguments

`pdata`	PolyData with a `Z` column.
`breaks`	either a vector of cut points or a scalar denoting the number of intervals that `Z` is to be cut into.
`propName`	name of the new column to append to `pdata`.
`propVals`	vector of values to associate with `Z` breaks.

Details

This function acts like the `cut` function to produce PolyData suitable for the `polyProps` plotting argument (see `addLabels`, `addLines`, `addPoints`, `addPolys`, `addStipples`, `plotLines`, `plotMap`, `plotPoints`, and `plotPolys`). The `Z` column of `pdata` is equivalent to the data vector `x` of the `cut` function.

Value

PolyData with the same columns as `pdata` plus an additional column `propName`.

See Also

`addLabels`, `addLines`, `addPoints`, `addPolys`, `addStipples`, `plotLines`, `plotMap`, `plotPoints`, `plotPolys`, PolyData, PolySet.

Examples

```
#--- create a PolyData object
pd <- data.frame(PID=1:10, Z=1:10)

#--- using 3 intervals, create a column named `col' and populate it with
#--- the supplied values
makeProps(pdata = pd, breaks = 3, propName = "col",
          propVals = c(1:3))
```

`makeTopography` *Make Topography Data From Freely Available Online Data*

Description

Make topography data suitable for the `contour` and `contourLines` functions using freely available global seafloor topography data.

Usage

```
makeTopography (dat, digits=2, func=NULL)
```

Arguments

`dat`	data frame with three optionally-named columns: `X`, `Y`, and `Z`. The columns must appear in that order.
`digits`	integer indicating the precision to be used by the function `round` on (X,Y) values.
`func`	function to summarize Z if (X,Y) points are duplicated. Defaults to `mean()` if no function is specified.

Details

Data obtained through the acquisition form at `http://topex.ucsd.edu/cgi-bin/get_data.cgi` is suitable for this function. `read.table` will import its ASCII files into R/S, creating the `data` argument for this function.

When creating data for regions with longitude values spanning -180° to 0°, consider subtracting 360 from the result's X coordinates (`x`).

When creating bathymetry data, consider negating the result's elevations (`z`) to give depths positive values.

Combinations of (X,Y) do not need to be complete (`z[x,y]=NA`) or unique (`z[x,y]=func(Z[x,y])`).

Value

List with elements `x`, `y`, and `z`. `x` and `y` are vectors, while `z` is a matrix with rownames `x` and colnames `y`. `contour` and `contourLines` expect data conforming to this list format.

See Also

`graphics::contour`, `grDevices::contourLines`, `convCP`.

Examples

```
#--- Example 1: Sample data frame and conversion.
file <- data.frame(X=c(1,1,2,2),Y=c(3,4,3,4),Z=c(5,6,7,8))
print(makeTopography(file));

#--- Example 2: Aleutian Islands bathymetry
require(PBSmapping);
isob <- c(100,500,1000,2500,5000);
icol <- rgb(0,0,seq(255,100,len=length(isob)),max=255);
afile <- paste(system.file(package="PBSmapping"),
"/Extra/aleutian.txt",sep="")
aleutian <- read.table(afile, header=FALSE,
   col.names=c("x","y","z"))
```

```
aleutian$x <- aleutian$x - 360
aleutian$z <- -aleutian$z
alBathy <- makeTopography(aleutian)
alCL <- contourLines(alBathy,levels=isob)
alCP <- convCP(alCL)
alPoly <- alCP$PolySet
attr(alPoly,"projection") <- "LL"
plotMap(alPoly,type="n");
addLines(alPoly,col=icol);
data(nepacLL); addPolys(nepacLL,col="gold");
legend(x="topleft",bty="n",col=icol,lwd=2,
   legend=as.character(isob));
```

`nepacLL` *Data: Shoreline of the NE Pacific Ocean (Normal Resolution)*

Description

PolySet of polygons for the northeast Pacific Ocean shoreline.

Usage

```
data(nepacLL)
```

Format

Data frame consisting of 4 columns: `PID` = primary polygon ID, `POS` = position of each vertex within a given polygon, `X` = longitude coordinate, and `Y` = latitude coordinate. Attributes: `projection = "LL"`.

Note

In R, the data must be loaded using the `data` function.

Source

Polygon data from the GSHHS (Global Self-consistent, Hierarchical, High-resolution Shoreline) database `gshhs_h.b`. Download from http://www.soest.hawaii.edu/wessel/gshhs/gshhs.html

```
nepacLL <-importGSHHS("gshhs_h.b", xlim=c(-190,-110), ylim=c(34,72),
                      level=1, n=15, xoff=-360)
```

References

Wessel, P. and Smith, W.H.F. (1996) A global, self-consistent, hierarchical, high-resolution shoreline database. *Journal of Geophysical Research* **101**, 8741–8743.
http://www.soest.hawaii.edu/pwessel/pwessel_pubs.html

See Also

Data: `nepacLLhigh`, `worldLL`, `worldLLhigh`, `bcBathymetry`

`importGSHHS`, `addPolys`, `clipPolys`, `plotPolys`, `plotMap`, `thickenPolys`, `thinPolys`

nepacLLhigh *Data: Shoreline of the NE Pacific Ocean (High Resolution)*

Description

PolySet of polygons for the northeast Pacific Ocean shoreline.

Usage

```
data(nepacLLhigh)
```

Format

Data frame consisting of 4 columns: PID = primary polygon ID, POS = position of each vertex within a given polygon, X = longitude coordinate, and Y = latitude coordinate. Attributes: projection = "LL".

Note

In R, the data must be loaded using the data function.

Source

Polygon data from the GSHHS (Global Self-consistent, Hierarchical, High-resolution Shoreline) database gshhs_f.b. Download from http://www.soest.hawaii.edu/wessel/gshhs/gshhs.html

```
nepacLLhigh <-importGSHHS("gshhs_f.b", xlim=c(-190,-110),
                 ylim=c(34,72), level=1, n=0, xoff=-360)
nepacLLhigh <- thinPolys(nepacLLhigh, tol=0.1, filter=3)
```

References

Wessel, P. and Smith, W.H.F. (1996) A global, self-consistent, hierarchical, high-resolution shoreline database. *Journal of Geophysical Research* **101**, 8741–8743.
http://www.soest.hawaii.edu/pwessel/pwessel_pubs.html

See Also

Data: nepacLL, worldLL, worldLLhigh, bcBathymetry

importGSHHS, addPolys, clipPolys, plotPolys, plotMap, thickenPolys, thinPolys

PBSmapping *PBS Mapping: Draw Maps and Implement Other GIS Procedures*

Description

This software has evolved from fisheries research conducted at the Pacific Biological Station (PBS) in Nanaimo, British Columbia, Canada. It extends the R language to include two-dimensional plotting features similar to those commonly available in a Geographic Information System (GIS). Embedded C code speeds algorithms from computational geometry, such as finding polygons that contain specified point events or converting between longitude-latitude and Universal Transverse Mercator (UTM) coordinates. It includes data for a global shoreline and other data sets in the public domain.

For a complete user's guide, see the file `PBSmapping-UG.pdf` in the R directory `.../library/PBSmapping/doc`.

`PBSmapping` includes 10 demos that appear as figures in the User's Guide. To see them, run the function `.PBSfigs()`.

More generally, a user can view all demos available from locally installed packages with the function `runDemos()` in our related (and recommended) package `PBSmodelling`.

PBSprint *Specify Whether to Print Summaries*

Description

Specify whether PBS Mapping should print object summaries or not. If not, data objects are displayed as normal.

Usage

```
PBSprint
```

Details

If `PBSprint = TRUE`, the mapping software will print summaries rather than the data frames for EventData, LocationSet, PolyData, and PolySet objects. If `PBSprint = FALSE`, it will print the data frames.

This variable's default value is `FALSE`.

Value

`TRUE` or `FALSE`, depending on the user's preference.

See Also

`summary`.

placeHoles *Place Holes Under Correct Solids*

Description

Place secondary polygons (SIDs) identified as holes (counter-clockwise rotation) under SIDs identified as solids (clockwise rotation) if the vertices of the holes lie completely within the vertices of the solids. This operation is performed for each primary polygon (PID).

Usage

```
placeHoles(polyset, minVerts=3)
```

Arguments

polyset	a valid **PBSmapping** PolySet.
minVerts	minimum number of vertices required for a polygon representing a hole to be retained (does not affect solids).

Details

The algorith identifies the rotation of each polygon down to the SID level using the **PBSmapping** function .calcOrientation, where output values of 1 = solids (clockwise rotation) and −1 = holes (counter-clockwise rotation). Then for each solid, the function tests whether each hole occurs within the solid. To facilitate computation, the algorithm assumes that once a hole is located in a solid, it will not occur in any other solid. This means that for each successive solid, the number of candidate holes will either decrease or stay the same.

This function makes use of the point.in.polygon function contained in the package **sp**. For each hole vertex, the latter algorithm returns a numeric value:
0 = hole vertex is strictly exterior to the solid;
1 = hole vertex is strictly interior to the solid;
2 = hole vertex lies on the relative interior of an edge of the solid;
3 = hole vertex coincides with a solid vertex.

Value

Returns the input PolySet where SID holes have been arranged beneath appropriate SID solids for each PID.

Author(s)

Rowan Haigh, Pacific Biological Station, Fisheries and Oceans Canada, Nanaimo BC.

References

See copyright notice in point.in.polygon.

See Also

importShapefile, point.in.polygon

plotLines *Plot a PolySet as Polylines*

Description

Plot a PolySet as polylines.

Usage

```
plotLines (polys, xlim = NULL, ylim = NULL, projection = FALSE,
           plt = c(0.11, 0.98, 0.12, 0.88), polyProps = NULL,
           lty = NULL, col = NULL, bg = 0, axes = TRUE,
           tckLab = TRUE, tck = 0.014, tckMinor = 0.5 * tck, ...)
```

Arguments

polys	PolySet to plot (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
projection	desired projection when PolySet lacks a projection attribute; one of "LL", "UTM", or a numeric value. If Boolean, specifies whether to check polys for a projection attribute.
plt	four element numeric vector (x1, x2, y1, y2) giving the coordinates of the plot region measured as a fraction of the figure region. Set to NULL if mai in par is desired.
polyProps	PolyData specifying which polylines to plot and their properties. par parameters passed as direct arguments supersede these data.
lty	vector describing line types (cycled by PID).
col	vector describing colours (cycled by PID).
bg	background colour of the plot.
axes	Boolean value; if TRUE, plot axes.
tckLab	Boolean vector (length 1 or 2); if TRUE, label the major tick marks. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
tck	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. If tckLab = TRUE, these tick marks will be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
tckMinor	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. These tick marks can not be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
...	additional par parameters, or the arguments main, sub, xlab, or ylab for the title function.

Details

This function plots a PolySet, where each unique (PID, SID) describes a polyline. It does not connect each polyline's last vertex to its first. Unlike plotMap, the function ignores the aspect ratio. It clips polys to xlim and ylim before plotting.

The function creates a blank plot when polys equals NULL. In this case, the user must supply both xlim and ylim arguments. Alternatively, it accepts the argument type = "n" as part of ..., which is equivalent to specifying polys = NULL, but requires a PolySet. In both cases, the function's behaviour changes slightly. To resemble the plot function, it plots the border, labels, and other parts according to par parameters such as col.

For additional help on the arguments lty and col, please see par.

Value

PolyData consisting of the PolyProps used to create the plot.

Note

To satisfy the aspect ratio, this plotting routine resizes the plot region. Consequently, par parameters such as plt, mai, and mar will change. When the function terminates, these changes persist to allow for additions to the plot.

See Also

addLines, calcLength, clipLines, closePolys, convLP, fixBound, fixPOS, locatePolys, thinPolys, thickenPolys.

Examples

```
#--- create a PolySet to plot
polys <- data.frame(PID=rep(1,4),POS=1:4,X=c(0,1,1,0),Y=c(0,0,1,1))
#--- plot the PolySet
plotLines(polys, xlim=c(-.5,1.5), ylim=c(-.5,1.5))
```

---

plotMap *Plot a PolySet as a Map*

Description

Plot a PolySet as a map, using the correct aspect ratio.

Usage

```
plotMap (polys, xlim = NULL, ylim = NULL, projection = TRUE,
         plt = c(0.11, 0.98, 0.12, 0.88), polyProps = NULL,
         border = NULL, lty = NULL, col = NULL, colHoles = NULL,
         density = NA, angle = NULL, bg = 0, axes = TRUE,
         tckLab = TRUE, tck = 0.014, tckMinor = 0.5 * tck, ...)
```

Arguments

polys	PolySet to plot (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
projection	desired projection when PolySet lacks a projection attribute; one of "LL", "UTM", or a numeric value. If Boolean, specifies whether to check polys for a projection attribute.
plt	four element numeric vector (x1, x2, y1, y2) giving the coordinates of the plot region measured as a fraction of the figure region. Set to NULL if mai in par is desired.
polyProps	PolyData specifying which polygons to plot and their properties. par parameters passed as direct arguments supersede these data.
border	vector describing edge colours (cycled by PID).
lty	vector describing line types (cycled by PID).
col	vector describing fill colours (cycled by PID).
colHoles	vector describing hole colours (cycled by PID). The default, NULL, should be used in most cases as it renders holes transparent. colHoles is designed solely to eliminate retrace lines when images are converted to PDF format. If colHoles is specified, underlying information (i.e., previously plotted shapes) will be obliterated. If NA is specified, only outer polygons are drawn, consequently filling holes.
density	vector describing shading line densities (lines per inch, cycled by PID).
angle	vector describing shading line angles (degrees, cycled by PID).
bg	background colour of the plot.
axes	Boolean value; if TRUE, plot axes.
tckLab	Boolean vector (length 1 or 2); if TRUE, label the major tick marks. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
tck	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. If tckLab = TRUE, these tick marks will be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
tckMinor	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. These tick marks can not be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
...	additional par parameters, or the arguments main, sub, xlab, or ylab for the title function.

Details

This function plots a PolySet, where each unique (PID, SID) describes a polygon. It connects each polygon's last vertex to its first. The function supports both borders (border, lty) and fills (col, density, angle). When supplied with the appropriate arguments, it can draw only borders or only fills . Unlike plotLines and plotPolys, it uses the aspect ratio supplied in the projection attribute of polys. If this attribute is missing, it attempts to use its projection argument. In the absence of both, it uses a default aspect ratio of 1:1. It clips polys to xlim and ylim before plotting.

The function creates a blank plot when polys equals NULL. In this case, the user must supply both xlim and ylim arguments. Alternatively, it accepts the argument type = "n" as part of ..., which is equivalent to specifying polys = NULL, but requires a PolySet. In both cases, the function's behaviour changes slightly. To resemble the plot function, it plots the border, labels, and other parts according to par parameters such as col.

For additional help on the arguments border, lty, col, density, and angle, please see polygon and par.

Value

PolyData consisting of the PolyProps used to create the plot.

Note

To satisfy the aspect ratio, this plotting routine resizes the plot region. Consequently, par parameters such as plt, mai, and mar will change. When the function terminates, these changes persist to allow for additions to the plot.

Author(s)

Nicholas Boers, Dept. of Computer Science, Grant MacEwan University, Edmonton AB

See Also

addLabels, addPolys, addStipples, clipPolys, closePolys, fixBound, fixPOS, locatePolys, plotLines, plotPoints, thinPolys, thickenPolys.

Examples

```
#--- create a PolySet to plot
polys <- data.frame(PID=rep(1,4),POS=1:4,X=c(0,1,1,0),Y=c(0,0,1,1))
#--- plot the PolySet
plotMap(polys,xlim=c(-.5,1.5),ylim=c(-.5,1.5),density=0,projection=1)
```

---

plotPoints *Plot EventData/PolyData as Points*

Description

Plot EventData/PolyData, where each unique EID or (PID, SID) describes a point.

Usage

```
plotPoints (data, xlim = NULL, ylim = NULL, projection = FALSE,
            plt = c(0.11, 0.98, 0.12, 0.88), polyProps = NULL,
            cex = NULL, col = NULL, pch = NULL, axes = TRUE,
            tckLab = TRUE, tck = 0.014, tckMinor = 0.5 * tck, ...)
```

Arguments

data	EventData or PolyData to plot (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.
projection	desired projection when PolySet lacks a projection attribute; one of "LL", "UTM", or a numeric value. If Boolean, specifies whether to check polys for a projection attribute.
plt	four element numeric vector (x1, x2, y1, y2) giving the coordinates of the plot region measured as a fraction of the figure region. Set to NULL if mai in par is desired.
polyProps	PolyData specifying which points to plot and their properties. par parameters passed as direct arguments supersede these data.
cex	vector describing character expansion factors (cycled by EID or PID).

col	vector describing colours (cycled by EID or PID).
pch	vector describing plotting characters (cycled by EID or PID).
axes	Boolean value; if TRUE, plot axes.
tckLab	Boolean vector (length 1 or 2); if TRUE, label the major tick marks. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
tck	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. If tckLab = TRUE, these tick marks will be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
tckMinor	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. These tick marks can not be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
...	additional par parameters, or the arguments main, sub, xlab, or ylab for the title function.

Details

This function clips data to xlim and ylim before plotting. It only adds PolyData containing X and Y columns.

The function creates a blank plot when polys equals NULL. In this case, the user must supply both xlim and ylim arguments. Alternatively, it accepts the argument type = "n" as part of ..., which is equivalent to specifying polys = NULL, but requires a PolySet. In both cases, the function's behaviour changes slightly. To resemble the plot function, it plots the border, labels, and other parts according to par parameters such as col.

For additional help on the arguments cex, col, and pch, please see par.

Value

PolyData consisting of the PolyProps used to create the plot.

Note

To satisfy the aspect ratio, this plotting routine resizes the plot region. Consequently, par parameters such as plt, mai, and mar will change. When the function terminates, these changes persist to allow for additions to the plot.

See Also

addPoints, combineEvents, convDP, findPolys, locateEvents.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R")) {
  data(nepacLL)
  data(surveyData)
}
#--- plot a map
plotMap(nepacLL, xlim=c(-136, -125), ylim=c(48, 57))
#--- add events
addPoints(surveyData, col=1:7)
```

`plotPolys` *Plot a PolySet as Polygons*

Description

Plot a PolySet as polygons.

Usage

```
plotPolys (polys, xlim = NULL, ylim = NULL, projection = FALSE,
           plt = c(0.11, 0.98, 0.12, 0.88), polyProps = NULL,
           border = NULL, lty = NULL, col = NULL, colHoles = NULL,
           density = NA, angle = NULL, bg = 0, axes = TRUE,
           tckLab = TRUE, tck = 0.014, tckMinor = 0.5 * tck, ...)
```

Arguments

`polys`	PolySet to plot (*required*).
`xlim`	range of X-coordinates.
`ylim`	range of Y-coordinates.
`projection`	desired projection when PolySet lacks a `projection` attribute; one of `"LL"`, `"UTM"`, or a numeric value. If Boolean, specifies whether to check `polys` for a `projection` attribute.
`plt`	four element numeric vector `(x1, x2, y1, y2)` giving the coordinates of the plot region measured as a fraction of the figure region. Set to `NULL` if `mai` in `par` is desired.
`polyProps`	PolyData specifying which polygons to plot and their properties. `par` parameters passed as direct arguments supersede these data.
`border`	vector describing edge colours (cycled by `PID`).
`lty`	vector describing line types (cycled by `PID`).
`col`	vector describing fill colours (cycled by `PID`).
`colHoles`	vector describing hole colours (cycled by `PID`). The default, `NULL`, should be used in most cases as it renders holes transparent. `colHoles` is designed solely to eliminate retrace lines when images are converted to PDF format. If `colHoles` is specified, underlying information (i.e., previously plotted shapes) will be obliterated. If `NA` is specified, only outer polygons are drawn, consequently filling holes.
`density`	vector describing shading line densities (lines per inch, cycled by `PID`).
`angle`	vector describing shading line angles (degrees, cycled by `PID`).
`bg`	background colour of the plot.
`axes`	Boolean value; if `TRUE`, plot axes.
`tckLab`	Boolean vector (length 1 or 2); if `TRUE`, label the major tick marks. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
`tck`	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. If `tckLab = TRUE`, these tick marks will be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.
`tckMinor`	numeric vector (length 1 or 2) describing the length of tick marks as a fraction of the smallest dimension. These tick marks can not be automatically labelled. If given a two-element vector, the first element describes the tick marks on the x-axis and the second element describes those on the y-axis.

...	additional par parameters, or the arguments main, sub, xlab, or ylab for the title function.

Details

This function plots a PolySet, where each unique (PID, SID) describes a polygon. It connects each polygon's last vertex to its first. The function supports both borders (border, lty) and fills (col, density, angle). When supplied with the appropriate arguments, it can draw only borders or only fills. Unlike plotMap, it ignores the aspect ratio. It clips polys to xlim and ylim before plotting.

This function creates a blank plot when polys equals NULL. In this case, the user must supply both xlim and ylim arguments. Alternatively, it accepts the argument type = "n" as part of ..., which is equivalent to specifying polys = NULL, but requires a PolySet. In both cases, the function's behaviour changes slightly. To resemble the plot function, it plots the border, labels, and other parts according to par parameters such as col.

For additional help on the arguments border, lty, col, density, and angle, please see polygon and par.

Value

PolyData consisting of the PolyProps used to create the plot.

Note

To satisfy the aspect ratio, this plotting routine resizes the plot region. Consequently, par parameters such as plt, mai, and mar will change. When the function terminates, these changes persist to allow for additions to the plot.

See Also

addLabels, addPolys, addStipples, clipPolys, closePolys, fixBound, fixPOS, locatePolys, plotLines, plotMap, plotPoints, thinPolys, thickenPolys.

Examples

```
#--- create a PolySet to plot
polys <- data.frame(PID=rep(1,4),POS=1:4,X=c(0,1,1,0),Y=c(0,0,1,1))
#--- plot the PolySet
plotPolys(polys, xlim=c(-.5,1.5), ylim=c(-.5,1.5), density=0)
```

PolyData *PolyData Objects*

Description

PBS Mapping functions that expect PolyData will accept properly formatted data frames in their place (see 'Details').

as.PolyData attempts to coerce a data frame to an object with class PolyData.

is.PolyData returns TRUE if its argument is of class PolyData.

Usage

```
as.PolyData(x, projection = NULL, zone = NULL)
is.PolyData(x, fullValidation = TRUE)
```

Arguments

x	data frame to be coerced or tested.
projection	optional projection attribute to add to PolyData, possibly overwriting an existing attribute.
zone	optional zone attribute to add to PolyData, possibly overwriting an existing attribute.
fullValidation	Boolean value; if TRUE, fully test x.

Details

We define PolyData as a data frame with a first column named PID and (optionally) a second column named SID. Unlike a PolySet, where each contour has many records corresponding to the vertices, a PolyData object must have only one record for each PID or each (PID, SID) combination. Conceptually, this object associates data with contours, where the data correspond to additional fields in the data frame. The R/S language conveniently allows data frames to contain fields of various atomic modes ("logical", "numeric", "complex", "character", and "null"). For example, PolyData with the fields (PID, PName) might assign character names to a set of primary polygons. Additionally, if fields X and Y exist (perhaps representing locations for placing labels), consider adding attributes zone and projection. Inserting the string "PolyData" as the class attribute's first element alters the behaviour of some functions, including print (if PBSprint is TRUE) and summary.

Our software particularly uses PolyData to set various plotting characteristics. Consistent with graphical parameters used by the R/S functions lines and polygon, column names can specify graphical properties:

- lty - line type in drawing the border and/or shading lines;
- col - line or fill colour;
- border - border colour;
- density - density of shading lines;
- angle - angle of shading lines.

When drawing polylines (as opposed to closed polygons), only lty and col have meaning.

Value

The as.PolyData method returns an object with classes "PolyData" and "data.frame", in that order.

See Also

EventData, LocationSet, PolySet.

PolySet *PolySet Objects*

Description

PBS Mapping functions that expect PolySet's will accept properly formatted data frames in their place (see 'Details').

as.PolySet attempts to coerce a data frame to an object with class PolySet.

is.PolySet returns TRUE if its argument is of class PolySet.

Usage

```
as.PolySet(x, projection = NULL, zone = NULL)
is.PolySet(x, fullValidation = TRUE)
```

Arguments

x	data frame to be coerced or tested.
projection	optional projection attribute to add to the PolySet, possibly overwriting an existing attribute.
zone	optional zone attribute to add to the PolySet, possibly overwriting an existing attribute.
fullValidation	Boolean value; if TRUE, fully test x.

Details

In our software, a PolySet data frame defines a collection of polygonal contours (i.e., line segments joined at vertices), based on four or five numerical fields:

- PID - the primary identification number for a contour;
- SID - optional, the secondary identification number for a contour;
- POS - the position number associated with a vertex;
- X - the horizontal coordinate at a vertex;
- Y - the vertical coordinate at a vertex.

The simplest PolySet lacks an SID column, and each PID corresponds to a different contour. By analogy with a child's "follow the dots" game, the POS field enumerates the vertices to be connected by straight lines. Coordinates (X, Y) specify the location of each vertex. Thus, in familiar mathematical notation, a contour consists of n points (x_i, y_i) with $i = 1, ..., n$, where i corresponds to the POS index. A PolySet has two potential interpretations. The first associates a line segment with each successive pair of points from 1 to n, giving a *polyline* (in GIS terminology) composed of the sequential segments. The second includes a final line segment joining points n and 1, thus giving a *polygon*.

The secondary ID field allows us to define regions as composites of polygons. From this point of view, each primary ID identifies a collection of polygons distinguished by secondary IDs. For example, a single management area (PID) might consist of two fishing areas, each defined by a unique SID. A secondary polygon can also correspond to an inner boundary, like the hole in a doughnut. We adopt the convention that POS goes from 1 to n along an outer boundary, but from n to 1 along an inner boundary, regardless of rotational direction. This contrasts with other GIS software, such as ArcView (ESRI 1996), in which outer and inner boundaries correspond to clockwise and counter-clockwise directions, respectively.

The SID field in a PolySet with secondary IDs must have integer values that appear in ascending order for a given PID. Furthermore, inner boundaries must follow the outer boundary that encloses them. The POS field for each contour (PID, SID) must similarly appear as integers in strictly increasing or decreasing order, for outer and inner boundaries respectively. If the POS field erroneously contains floating-point numbers, fixPOS can renumber them as sequential integers, thus simplifying the insertion of a new point, such as point 3.5 between points 3 and 4.

A PolySet can have a projection attribute, which may be missing, that specifies a map projection. In the current version of PBS Mapping, projection can have character values "LL" or "UTM", referring to "Longitude-Latitude" and "Universal Transverse Mercator". We explain these projections more completely below. If projection is numeric, it specifies the aspect ratio r, the number of x units per y unit. Thus, r units of x on the graph occupy the same distance as one unit of y. Another optional attribute zone specifies the UTM zone (if projection="UTM") or the preferred zone for conversion from Longitude-Latitude (if projection="LL").

A data frame's class attribute by default contains the string "data.frame". Inserting the string "PolySet" as the class vector's first element alters the behaviour of some functions. For example, the summary function

will print details specific to a PolySet. Also, when PBSprint is TRUE, the print function will display a PolySet's summary rather than the contents of the data frame.

Value

The as.PolySet method returns an object with classes "PolySet" and "data.frame", in that order.

References

Environmental Systems Research Institute (ESRI). (1996) *ArcView GIS: The Geographic Information System for Everyone*. ESRI Press, Redlands, California. 340 pp.

See Also

EventData, LocationSet, PolyData.

---

print *Print PBS Mapping Objects*

Description

This function displays information about a PBS Mapping object.

summary.EventData, summary.LocationSet, summary.PolyData, and summary.PolySet produce an object with class summary.PBS.

Usage

```
## S3 method for class 'EventData'
print(x, ...)
## S3 method for class 'LocationSet'
print(x, ...)
## S3 method for class 'PolyData'
print(x, ...)
## S3 method for class 'PolySet'
print(x, ...)
## S3 method for class 'summary.PBS'
print(x, ...)
```

Arguments

x	a PBS Mapping object of appropriate class.
...	additional arguments to print.

See Also

EventData, LocationSet, PBSprint, PolyData, PolySet, summary.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- change to summary printing style
PBSprint <- TRUE
#--- print the PolySet
print(nepacLL)
```

---

pythagoras *Data: Pythagoras' Theorem Diagram PolySet*

Description

PolySet of shapes to prove Pythagoras' Theorem: $a^2 + b^2 = c^2$.

Usage

```
data(pythagoras)
```

Format

4 column data frame: `PID` = primary polygon ID, `POS` = position of each vertex within a given polyline, `X` = X-coordinate, and `Y` = Y-coordinate. Attributes: `projection = 1`.

Note

In R, the data must be loaded using the `data` function.

Source

An artificial construct to illustrate the proof of Pythagoras' Theorem using trigonometry.

See Also

`addPolys`, `plotPolys`, `plotMap`, PolySet.

---

refocusWorld *Refocus the worldLL/worldLLhigh Data Sets*

Description

Refocus the `worldLL`/`worldLLhigh` data sets, e.g., refocus them so that Eastern Canada appears to the west of Western Europe.

Usage

```
refocusWorld (polys, xlim = NULL, ylim = NULL)
```

Arguments

polys	PolySet with one or more polygons; typically worldLL or worldLLhigh (*required*).
xlim	range of X-coordinates.
ylim	range of Y-coordinates.

Details

This function accepts a PolySet containing one or more polygons with X-coordinates that collectively span approximately 360 degrees. The function effectively joins the PolySet into a cylinder and then splits it at an arbitrary longitude according to the user-specified limits. Modifications in the resulting PolySet are restricted to shifting X-coordinates by +/- multiples of 360 degrees, and instead of clipping polygons, the return value simply omits out-of-range polygons.

Value

PolySet, likely a subset of the input PolySet, which retains the same PID/SID values.

Author(s)

Nicholas Boers, Dept. of Computer Science, Grant MacEwan University, Edmonton AB

See Also

joinPolys

Examples

```
#--- load appropriate data
data(worldLL)
#--- set limits
xlim <- c(-100,25)
ylim <- c(0,90)
#--- refocus and plot the world
polys <- refocusWorld(worldLL, xlim, ylim)
plotMap(polys, xlim, ylim)
```

---

summary *Summarize PBS Mapping Objects*

Description

summary method for PBS Mapping classes.

Usage

```
## S3 method for class 'EventData'
summary(object, ...)
## S3 method for class 'LocationSet'
summary(object, ...)
## S3 method for class 'PolyData'
summary(object, ...)
## S3 method for class 'PolySet'
summary(object, ...)
```

Arguments

object	a PBS Mapping object, such as EventData, a LocationSet, PolyData, or a PolySet.
...	further arguments passed to or from other methods.

Details

After creating a list of summary statistics, this function assigns the class "summary.PBS" to the output in order to accomplish formatted printing via print.summary.PBS.

Value

A list of summary statistics.

See Also

EventData, LocationSet, PBSprint, PolyData, PolySet.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(surveyData)
print(summary(surveyData))
```

---

surveyData *Data: Tow Information from Pacific Ocean Perch Survey*

Description

EventData of Pacific ocean perch (POP) tow information (1966-89).

Usage

```
data(surveyData)
```

Format

Data frame consisting of 9 columns: PID = primary polygon ID, POS = position of each vertex within a given polygon, X = longitude coordinate, Y = latitude coordinate, trip = trip ID, tow = tow number in trip, catch = catch of POP (kg), effort = tow effort (minutes), depth = fishing depth (m), and year = year of survey trip. Attributes: projection = "LL", zone = 9.

Note

In R, the data must be loaded using the data function.

Source

The GFBio database, maintained at the Pacific Biological Station (Fisheries and Oceans Canada, Nanaimo, BC V9T 6N7), archives catches and related biological data from commercial groundfish fishing trips and research/assessment cruises off the west coast of British Columbia (BC).

The POP (*Sebastes alutus*) survey data were extracted from GFBio. The data extraction covers bottom trawl surveys that focus primarily on POP biomass estimation: 1966-89 for the central BC coast and 1970-85 for the west coast of Vancouver Island. Additionally, a 1989 cruise along the entire BC coast concentrated on the collection of biological samples. Schnute et al. (2001) provide a more comprehensive history of POP surveys including the subset of data presented here.

References

Schnute, J.T., Haigh, R., Krishka, B.A. and Starr, P. (2001) Pacific ocean perch assessment for the west coast of Canada in 2001. *Canadian Science Advisory Secretariat, Research Document* **2001/138**, 90 pp.

See Also

addPoints, combineEvents, EventData, findPolys, makeGrid, plotPoints.

---

thickenPolys *Thicken a PolySet of Polygons*

Description

Thicken a PolySet, where each unique (PID, SID) describes a polygon.

Usage

```
thickenPolys (polys, tol = 1, filter = 3, keepOrig = TRUE,
              close = TRUE)
```

Arguments

polys	PolySet to thicken.
tol	tolerance (in kilometres when proj is "LL" and "UTM"; otherwise, same units as polys).
filter	minimum number of vertices per result polygon.
keepOrig	Boolean value; if TRUE, keep the original points in the PolySet.
close	Boolean value; if TRUE, create intermediate vertices between each polygon's last and first vertex, if necessary.

Details

This function thickens each polygon within polys according to the input arguments.

If keepOrig = TRUE, all of the original vertices appear in the result. It calculates the distance between two sequential original vertices, and if that distance exceeds tol, it adds a sufficient number of vertices spaced evenly between the two original vertices so that the distance between vertices no longer exceeds tol. If close = TRUE, it adds intermediate vertices between the last and first vertices when necessary.

If keepOrig = FALSE, only the first vertex of each polygon is guaranteed to appear in the results. From this first vertex, the algorithm walks the polygon summing the distance between vertices. When this cumulative distance exceeds tol, it adds a vertex on the line segment under inspection. After doing so, it resets the distance sum, and walks the polygon from this new vertex. If close = TRUE, it will walk the line segment from the last vertex to the first.

Value

PolySet containing the thickened data. The function recalculates the POS values for each polygon.

See Also

thinPolys.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- plot Vancouver Island
plotMap(nepacLL[nepacLL$PID == 33, ])
#--- calculate a thickened version using a 30 kilometres tolerance,
#--- without keeping the original points
p <- thickenPolys(nepacLL[nepacLL$PID == 33, ], tol = 30, keepOrig = FALSE)
#--- convert the PolySet to EventData by dropping the PID column and
#--- renaming POS to EID
p <- p[-1];
names(p)[1] <- "EID";
#--- convert the now invalid PolySet into a data frame, and then into
#--- EventData
p <- as.EventData(as.data.frame(p), projection="LL");
#--- plot the results
addPoints(p, col=2, pch=19)
```

---

thinPolys *Thin a PolySet of Polygons*

Description

Thin a PolySet, where each unique (PID, SID) describes a polygon.

Usage

```
thinPolys (polys, tol = 1, filter = 3)
```

Arguments

polys	PolySet to thin.
tol	tolerance (in kilometres when proj is "LL" and "UTM"; otherwise, same units as polys).
filter	minimum number of vertices per result polygon.

Details

This function executes the Douglas-Peuker line simplification algorithm on each polygon within polys.

Value

PolySet containing the thinned data. The function recalculates the POS values for each polygon.

See Also

thickenPolys.

Examples

```
#--- load the data (if using R)
if (!is.null(version$language) && (version$language == "R"))
  data(nepacLL)
#--- plot a thinned version of Vancouver Island (3 km tolerance)
plotMap(thinPolys(nepacLL[nepacLL$PID == 33, ], tol = 3))
#--- add the original Vancouver Island in a different line type to
#--- emphasize the difference
addPolys(nepacLL[nepacLL$PID == 33, ], border=2, lty=8, density=0)
```

---

towData *Data: Tow Information from Longspine Thornyhead Survey*

Description

PolyData of tow information for a longspine thornyhead survey (2001).

Usage

```
data(towData)
```

Format

Data frame consisting of 8 columns: `PID` = primary polygon ID, `POS` = position of each vertex within a given polygon, `X` = longitude coordinate, `Y` = latitude coordinate, `depth` = fishing depth (m), `effort` = tow effort (minutes), `distance` = tow track distance (km), `catch` = catch of longspine thornyhead (kg), and `year` = year of survey. Attributes: `projection = "LL"`, `zone = 9`.

Note

In R, the data must be loaded using the `data` function.

Source

The GFBio database, maintained at the Pacific Biological Station (Fisheries and Oceans Canada, Nanaimo, BC V9T 6N7), archives catches and related biological data from commercial groundfish fishing trips and research/assessment cruises off the west coast of British Columbia (BC). The longspine thornyhead (*Sebastolobus altivelis*) survey data were extracted from GFBio. Information on the first 45 tows from the 2001 survey (Starr et al. 2002) are included here. Effort is time (minutes) from winch lock-up to winch release.

References

Starr, P.J., Krishka, B.A. and Choromanski, E.M. (2002) Trawl survey for thornyhead biomass estimation off the west coast of Vancouver Island, September 15 - October 2, 2001. *Canadian Technical Report of Fisheries and Aquatic Sciences* **2421**, 60 pp.

See Also

makeProps, PolyData, towTracks.

towTracks *Data: Tow Track Polylines from Longspine Thornyhead Survey*

Description

PolySet of geo-referenced polyline tow track data from a longspine thornyhead survey (2001).

Usage

```
data(towTracks)
```

Format

Data frame consisting of 4 columns: `PID` = primary polygon ID, `POS` = position of each vertex within a given polyline, `X` = longitude coordinate, and `Y` = latitude coordinate. Attributes: `projection = "LL"`, `zone = 9`.

Note

In R, the data must be loaded using the `data` function.

Source

The longspine thornyhead (*Sebastolobus altivelis*) tow track spatial coordinates are available at the Pacific Biological Station (Fisheries and Oceans Canada, Nanaimo, BC V9T 6N7). The geo-referenced coordinates of the first 45 tows from the 2001 survey (Starr et al. 2002) are included here. Coordinates are recorded once per minute between winch lock-up and winch release.

References

Starr, P.J., Krishka, B.A. and Choromanski, E.M. (2002) Trawl survey for thornyhead biomass estimation off the west coast of Vancouver Island, September 15 - October 2, 2001. *Canadian Technical Report of Fisheries and Aquatic Sciences* **2421**, 60 pp.

See Also

`addLines`, `calcLength`, `clipLines`, `plotLines`, PolySet, `towData`.

worldLL *Data: Shorelines of the World (Normal Resolution)*

Description

PolySet of polygons for the global shorelines.

Usage

```
data(worldLL)
```

Format

Data frame consisting of 4 columns: `PID` = primary polygon ID, `POS` = position of each vertex within a given polygon, `X` = longitude coordinate, and `Y` = latitude coordinate. Attributes: `projection = "LL"`.

Note

In R, the data must be loaded using the data function.

Source

Polygon data from the GSHHS (Global Self-consistent, Hierarchical, High-resolution Shoreline) database gshhs_l.b. Download from http://www.soest.hawaii.edu/wessel/gshhs/gshhs.html

```
worldLL <-importGSHHS("gshhs_l.b", xlim=c(-20,360), ylim=c(-90,90),
                      level=1, n=15, xoff=0)
worldLL <- .fixGSHHSWorld(worldLL)
```

References

Wessel, P. and Smith, W.H.F. (1996) A global, self-consistent, hierarchical, high-resolution shoreline database. *Journal of Geophysical Research* **101**, 8741–8743.
http://www.soest.hawaii.edu/pwessel/pwessel_pubs.html

See Also

Data: worldLLhigh, nepacLL, nepacLLhigh

importGSHHS, addPolys, clipPolys, plotPolys, plotMap, thickenPolys, thinPolys

---

worldLLhigh *Data: Shorelines of the World (High Resolution)*

Description

PolySet of polygons for the global shorelines.

Usage

```
data(worldLLhigh)
```

Format

Data frame consisting of 4 columns: PID = primary polygon ID, POS = position of each vertex within a given polygon, X = longitude coordinate, and Y = latitude coordinate. Attributes: projection = "LL".

Note

In R, the data must be loaded using the data function.

Source

Polygon data from the GSHHS (Global Self-consistent, Hierarchical, High-resolution Shoreline) database gshhs_i.b. Download from http://www.soest.hawaii.edu/wessel/gshhs/gshhs.html

```
worldLLhigh <-importGSHHS("gshhs_i.b", xlim=c(-20,360),
                  ylim=c(-90,90), level=1, n=15, xoff=0)
worldLLhigh <- .fixGSHHSWorld(worldLLhigh)
```

References

Wessel, P. and Smith, W.H.F. (1996) A global, self-consistent, hierarchical, high-resolution shoreline database. *Journal of Geophysical Research* **101**, 8741–8743.
http://www.soest.hawaii.edu/pwessel/pwessel_pubs.html

See Also

Data: worldLL, nepacLL, nepacLLhigh

importGSHHS, addPolys, clipPolys, plotPolys, plotMap, thickenPolys, thinPolys

Index

∗Topic **IO**
print, 102
∗Topic **aplot**
addBubbles, 47
addLabels, 48
addLines, 50
addPoints, 51
addPolys, 52
addStipples, 53
plotPoints, 96
∗Topic **classes**
EventData, 70
LocationSet, 85
PolyData, 99
PolySet, 100
∗Topic **datasets**
bcBathymetry, 55
nepacLL, 89
nepacLLhigh, 90
pythagoras, 103
surveyData, 105
towData, 108
towTracks, 109
worldLL, 109
worldLLhigh, 110
∗Topic **documentation**
EventData, 70
LocationSet, 85
PBSmapping, 91
PolyData, 99
PolySet, 100
∗Topic **file**
importEvents, 76
importGSHHS, 76
importLocs, 78
importPolys, 78
importShapefile, 79
∗Topic **hplot**
plotLines, 93
plotMap, 94
plotPolys, 98
∗Topic **iplot**
locateEvents, 83
locatePolys, 84
∗Topic **logic**
joinPolys, 82
∗Topic **manip**
appendPolys, 54
calcArea, 56
calcCentroid, 57
calcConvexHull, 57
calcLength, 58
calcMidRange, 59
calcSummary, 60
calcVoronoi, 61
clipLines, 62
clipPolys, 63
closePolys, 63
combineEvents, 64
combinePolys, 65
convCP, 66
convDP, 67
convLP, 68
convUL, 69
dividePolys, 70
extractPolyData, 71
findCells, 72
findPolys, 73
fixBound, 74
fixPOS, 75
isConvex, 80
isIntersecting, 81
joinPolys, 82
makeGrid, 86
makeProps, 87
makeTopography, 88
placeHoles, 92
refocusWorld, 103
thickenPolys, 106
thinPolys, 107
∗Topic **methods**
summary, 104
∗Topic **sysdata**
PBSprint, 91

addBubbles, 47
addLabels, 48, *52*, *87*, *96*, *99*
addLines, 50, *68*, *87*, *94*, *109*
addPoints, *49*, 51, *53*, *58*, *61*, *67*, *84*, *87*, *97*, *106*

addPolys, *48*, 52, *53, 54*, *58*, *61*, *83, 84*, *86, 87*, *89, 90*, *96*, *99*, *103*, *110, 111*
addStipples, *52*, 53, *87*, *96*, *99*
appendPolys, 54, *68*, *83, 84*
as.EventData *(EventData)*, 70
as.LocationSet *(LocationSet)*, 85
as.PolyData *(PolyData)*, 99
as.PolySet *(PolySet)*, 100

bcBathymetry, 55, *89, 90*

calcArea, 56, *57–61*
calcCentroid, *49*, *56*, 57, *58–61*
calcConvexHull, 57, *60, 61*
calcLength, *50*, *56, 57*, 58, *59, 60*, *94*, *109*
calcMidRange, *49*, *56–59*, 59, *60, 61*
calcSummary, *49*, *56–59*, 60, *61*
calcVoronoi, 61
clipLines, *50*, 62, *63*, *94*, *109*
clipPolys, *52*, *54*, *62*, 63, *83, 84*, *86*, *89, 90*, *96*, *99*, *110, 111*
closePolys, *50*, *52*, *54*, 63, *68, 69*, *74, 75*, *83, 84*, *94*, *96*, *99*
combineEvents, *51*, *60*, 64, *73, 74*, *84*, *86*, *97*, *106*
combinePolys, 65, *70*
contour, *55*, *66*, *88*
contourLines, *55*, *66*, *88*
convCP, *55*, 66, *68*, *88*
convDP, *51*, 67, *84*, *97*
convLP, *50*, *54*, *66*, 68, *94*
convUL, 69
cut, *87*

data, *55*, *89, 90*, *103*, *105*, *108–110*
dividePolys, *66*, 70

EventData, *47–49*, *51*, *65*, *67*, 70, *72, 73*, *79*, *83–85*, *96*, *100*, *102*, *105, 106*
extractPolyData, 71

findCells, *65*, 72, *74*, *84*, *86*
findPolys, *51*, *60*, *65*, *72, 73*, 73, *84–86*, *97*, *106*
fixBound, *50*, *52*, *54*, *62–64*, *69*, 74, *75*, *83*, *94*, *96*, *99*
fixPOS, *50*, *52*, *54*, *64*, *74*, 75, *83, 84*, *94*, *96*, *99*, *101*

importEvents, 76, *77–80*
importGSHHS, *76*, 76, *78–80*, *89, 90*, *110, 111*
importLocs, *76, 77*, 78, *79, 80*
importPolys, *76–78*, 78, *80*
importShapefile, *76–79*, 79, *92*
is.EventData *(EventData)*, 70
is.LocationSet *(LocationSet)*, 85
is.PolyData *(PolyData)*, 99
is.PolySet *(PolySet)*, 100
isConvex, *74, 75*, 80, *81*
isIntersecting, *74, 75*, *80*, 81

joinPolys, *54*, *68*, 82, *84*, *104*

legend, *48*
lines, *50*, *100*
locateEvents, *51*, *57, 58*, *60, 61*, *65*, *73, 74*, 83, *97*
locatePolys, *50*, *52*, *56, 57*, *59, 60*, *65*, *73, 74*, *83*, 84, *94*, *96*, *99*
LocationSet, *65*, *71–74*, 85, *100*, *102*, *105*
locator, *83, 84*

makeGrid, *60*, *65*, *72–74*, 86, *106*
makeProps, *60*, *65*, *72*, 87, *108*
makeTopography, *66*, 88
mean, *65*

na.omit, *83, 84*
nepacLL, *55*, 89, *90*, *110, 111*
nepacLLhigh, *55*, *89*, 90, *110, 111*

par, *48–53*, *83, 84*, *93–99*
PBSmapping, 91
PBSmapping-package *(PBSmapping)*, 91
PBSprint, *71*, *85*, 91, *100*, *102*, *105*
placeHoles, *80*, 92
plot, *94, 95*, *97*, *99*
plotLines, *50*, *52*, *68*, *87*, 93, *95, 96*, *99*, *109*
plotMap, *52–54*, *58*, *61*, *83, 84*, *87*, *89, 90*, *94*, 94, *99*, *103*, *110, 111*
plotPoints, *49*, *51–53*, *58*, *61*, *67*, *83, 84*, *87*, *96*, 96, *99*, *106*
plotPolys, *52–54*, *58*, *61*, *84*, *87*, *89, 90*, *95*, 98, *103*, *110, 111*
point.in.polygon, *80*, *92*
points, *51*, *53*
PolyData, *48–53*, *56, 57*, *59, 60*, *65–67*, *71, 72*, *79–81*, *84, 85*, *87*, *93–99*, 99, *102*, *105*, *108*
polygon, *52*, *95*, *99, 100*
PolySet, *49, 50*, *52–68*, *70–75*, *79–82*, *84–87*, *89, 90*, *93–100*, 100, *102–107*, *109, 110*
print, *71*, *85*, *100*, *102*, 102
print.summary.PBS, *105*
pythagoras, 103

read.table, *88*
refocusWorld, 103

sum, *65*
summary, *71*, *85*, *91*, *100–102*, 104
summary.EventData, *102*
summary.LocationSet, *102*

summary.PolyData, *102*
summary.PolySet, *102*
surveyData, *48*, 105

text, *49*
thickenPolys, *50*, *52*, *83, 84*, *86*, *89, 90*, *94*, *96*, *99*, 106, *108*, *110, 111*
thinPolys, *50*, *52*, *83, 84*, *89, 90*, *94*, *96*, *99*, *107*, 107, *110, 111*
title, *93*, *95*, *97*, *99*
towData, 108, *109*
towTracks, *108*, 109

worldLL, *89, 90*, 109, *111*
worldLLhigh, *89, 90*, *110*, 110