Processing Remote Sensing Data with Python

Ryan J. Dillon

Faculty of Life and Environmental Sciences
University of Iceland

Processing Remote Sensing Data with Python

Ryan J. Dillon

10 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Joint Nordic Masters Programme in Marine Ecosystems and Climate

Advisor / Faculty Representative

Guðrún Marteinsdóttir

Faculty of Life and Environmental Sciences
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, June 2013

Processing Remote Sensing Data with Python
10 ECTS thesis submitted in partial fulfillment of a *Magister Scientiarum* degree in Joint Nordic Masters Programme in Marine Ecosystems and Climate

Faculty of Life and Environmental Sciences
School of Engineering and Natural Sciences
University of Iceland
Askja, Sturlugata 7
101, Reykjavik
Iceland

Telephone: 525 4600

Bibliographic information:
Dillon, Ryan J., 2013, *Processing Remote Sensing Data with Python*, Independent Study, Faculty of Life and Environmental Sciences, University of Iceland

CONTENTS

CHAPTER

ONE

ACKNOWLEDGEMENTS

First, I would like to thank the American Scandinavian Foundation in their support of my work. I would also like to thank Guðrún Marteinsdóttir for her coordination and advising, and Thorvaldur Gunlaugsson and Mette Mauritzen for their advising on my project for which this coding is intended to support. Lastly, I am very appreciative toward Gisli Víkingsson and Hafrannsóknastofnun for arranging a place for me to work.

CHAPTER

TWO

INTRODUCTION

With public access available for numerous satellite imaging products, modelling in atmospheric and oceanographic applications has become increasingly more prevalent.

Though there are numerous tools available for geospatial development, their use is more commonly applied towards mapping applications. With this being the case, there are a number of valuable texts for using these tools in such mapping applications [11] [1]; though, documentation for processing of remote sensing datasets is limited to brief contributions on personal blogs or manual pages and tutorials specific to one library or dataset. Python programming methods for performing such tasks will be focused on here, collecting various code and information from these text, blogs, etc. and presenting original code that may be employed in scripts to perform commonly required tasks in processing remote sensing data.

For a general place to get started with geospatial work with Python, Erik Westra's "Python Geospatial Development" is an excellent resource, and it will provide an excellent overview of geospatial work with python before looking at specific remote sensing datasets.

2.1 Geosetup

The tools presented here are collected into a Python program repository(currently under development) that will produce standardized gridded satellite data files that may be used as input for various models. This repository is publicy available on the collaborative coding site GitHub.

Link to repository on GitHub: https://github.com/ryanjdillon/geosetup

Or, you may download the code as a zip-archive.

If you are interested in collaborating on this effort, please feel encouraged to fork the repository on GitHub and offer your contributions.

2.2 Improving this Documentation

There are many improvements that can always be made in either the code shown here, or the explanation of a particular approach to something. If you find something broken, or stuck with an inadequate explanation, feel free to contact me so that I might correct it.

Thanks, Ryan ryanjamesdillon@gmail.com

CHAPTER

THREE

BRIEF OVERVIEW OF COORDINATE SYSTEMS AND MAP PROJECTIONS

Before working with georeferenced datasets, it is important to understand how things can be referenced geographically.

Positions on earth's surface can be represented using different systems, the most accurate of these being **geodetic positions** determined by coordinates from a particular geographic coordinate system, or **geodetic system**.

In addition to coordinate systems, the assumed shape of the earth will affect how coordinate positions are translated to earth's surface. **Geodetic Datums** are mathematical definitions of for the shape of the earth, which in turn defines a geodetic system.

3.1 Coordinate Systems

Coordinate systems are categorized into two groups, both of which being commonly used in representing remote sensing data:

- **unprojected coordinate systems** these are 3-dimensional coordinate systems, such as latitude and longitude (referred to as *Geographic Coordinate System* in common GIS software)
- **projected coordinate systems** these are 2-dimensional, there are many all of which having an advantage over another for a particular use.

Either system may be used depending on the data format (e.g. *netCDF HDF* containing arrays of unprojected data vs. *GeoTIFF* images containing projected data). When working with multiple datasets, or doing some processing of a dataset, you may need to change between different coordinate systems.

Most often data will coincide with a latitude and longitude (i.e. unprojected). If it is desired to interpolate or re-sample this data, transforming the data to a projected system allows for simpler and a greater variety of methods to be used.

3.2 Earth's Shape and Datums

It is often easiest to make the assumption that the earth is a perfect sphere when working with spherical coordinates. However, the earth's shape is in fact an oblate ellipsoid, with its polar radius being approximately 21 km less then the equatorial radius [9].

This shape has been calculated using different systems for both the entire globe and for regional areas, known as *datums*.

The most widely used datum for Geographic Information Systems is the **World Geodetic System (WGS84)**, which is used by most Global Positions System (GPS) devices by default.

It is important to know which datum was used to record positions of data, as there can be sizable differences between actual physical position of points with the same coordinates, depending on the datum.

3.3 Map Projections

In order to better visualise the earths surface and simplify working with certain geographic information, 2-dimensional representations can be calculated using various mathematical transformations.

Depending on the size of the area of interest (i.e. a small coastal area vs. an entire ocean) different projections will provide the greatest accuracy when interpolation of data is necessary.

3.3.1 Mercator

The Mercator Projection is commonly used for general mapping applications where visualization is a priority over accuracy of size and shape near the poles. Variations of it are used for mapping applications, such as is used for Google Maps [15].

Though useful for many mapping applications, this projection should be avoided when interpolation of data is necessary due to increasing distortion error near the poles.

3.3.2 UTM

The Universal Transverse Mercator (UTM) projection is a variation of the Mercator projection, which uses a collection of different projections of 60 zones on the earths surface to allow accurate positioning and minimal distortion on a 2-dimensional projection [8].

Though this system provides a way of accurate approach to a Mercator, it becomes difficult to work with when the area of interest spans multiple zones.

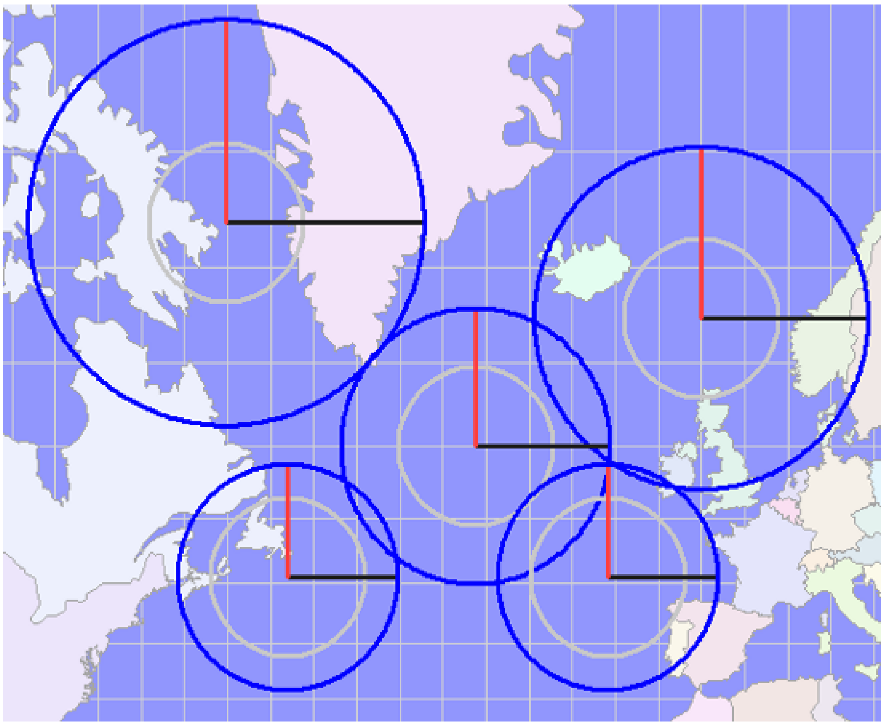


Figure 3.1: Mercator projection with Tissot circles showing increased distortion near the poles [5].

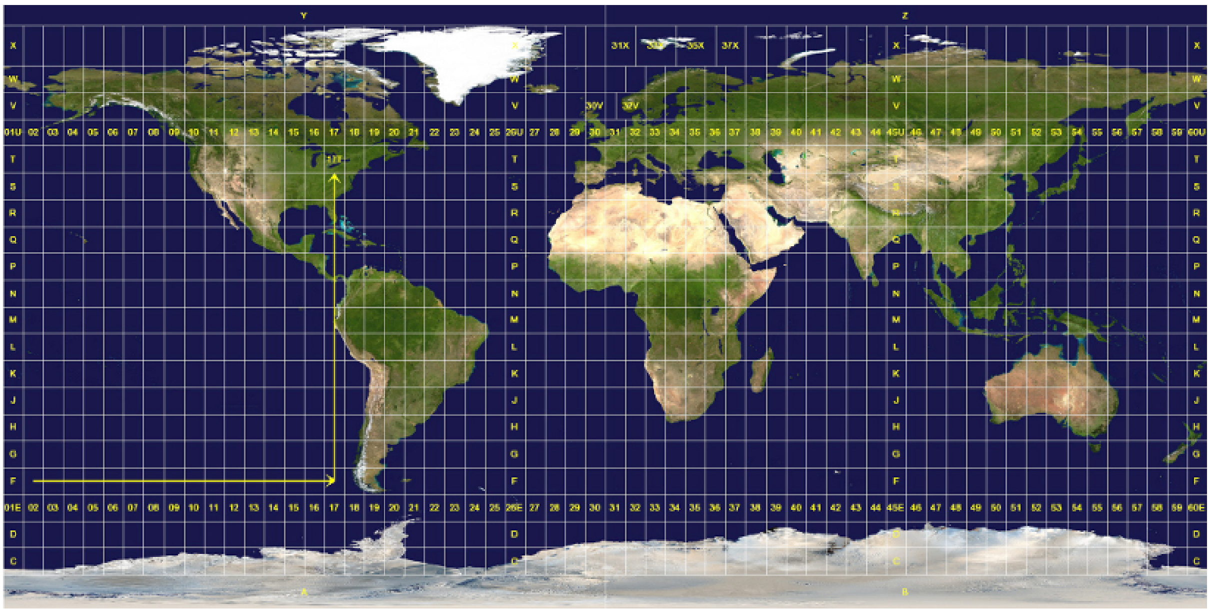


Figure 3.2: Universal Transverse Mercator (UTM) projection displaying the different zones [8]

3.3.3 Albers Equal-Area

The Albers equal-area conic projection is a projection that is useful where area needs to be preserved for large geographical areas. When working with data, as is needed when interpolating data over such an area.

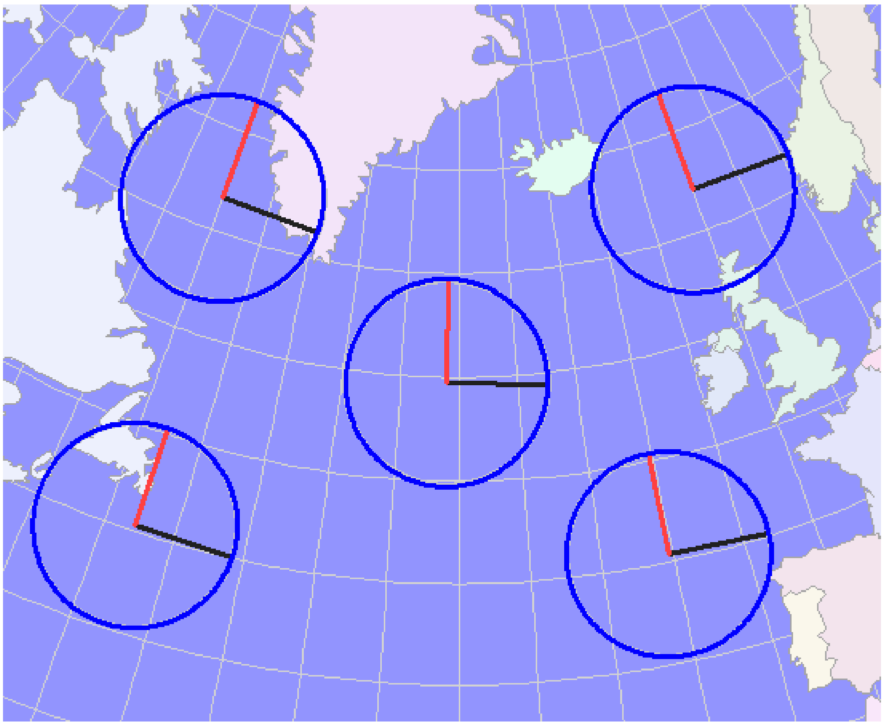


Figure 3.3: Albers Equal-Area Conic projection with Tissot circles [5].

CHAPTER

FOUR

GETTING STARTED WITH OCEAN COLOR AND BATHYMETRIC DATA

The datasets that will be focussed on are those containing parameters that are most generally applicable to marine ecosystem modeling: sea surface temperature, chlorophyll a, and bottom depth. The concepts and methods used for these may be extrapolated to other datasets, and should provide a good framework for processing such data.

4.1 Data Formats

4.1.1 netCDF HDF

The most common formats used for oceanographic and atmospheric datasets are Unidata's Network Common Data Form (netCDF) and HDF Group's Hierarchical Data Format (HDF) [18] [20].

Both of these formats have gone through multiple revisions, with different functionality and backwards compatibility being available between versions, with the latest versions being in most common use, netCDF-4 and HDF5 respectively.

As the name describes, HDF is hierarchical in its internal organization, which has many advantages particularly advanced organization of data and efficient storage and access to large amounts of data. HDF5 is the most current and widely used HDF format with many tools and libraries for accessing and creating datasets in this format.

4.1.2 GeoTIFF

Another common format for geospatial data is in the form of raster data, or data that is represented by grid cells particularly in the form of an image, such as GeoTIFF files.

The GeoTIFF format is a standardized file format, which is a modification of the Tagged Image File Format (TIFF) specially suited for storing datasets referenced in a projected coordinate system. Each data value will correspond to a grid cell (i.e. pixel) of the image, which is saved as a *tag* as metadata together with the image data [10]. The images pixel corresponds to an area enclosed by geographic boundaries defined by the projection used to create that GeoTIFF.

4.1.3 Arc/Info ASCII Grid and Binary Grid

Due to the popularity of ESRI's ArcGIS software, another format commonly seen is Arc/Info ASCII grid format and binary grid format. Developed by ESRI for their ArcGIS software, both formats store data in a similar way to GeoTIFF in that they relate a set of data values to grid cell areas, defined by some geographic bounds.

The binary format is a proprietary format that was developed to add functionality and prevent unlicensed use of data produced by their software.

The Arc/Info ASCII Grid is a simple ASCII text file format that is still found in used by some due to its simple non-proprietary format. The ASCII format includes a series of header rows defining the rows and columns in the grid, position of the center and corners, cell size, fill value for missing data, and then space-delimited values. As described by the Oak Ridge National Laboratory site [17]:

```
<NCOLS xxx>
<NROWS xxx>
<XLLCENTER xxx | XLLCORNER xxx>
<YLLCENTER xxx | YLLCORNER xxx>
<CELLSIZE xxx>
{NODATA_VALUE xxx}
row 1
row 2
.
.
.
row n
```

Note: There are many other raster image formats besides GeoTIFF and Arc/Info grid formats that geospatial data can be stored in. The GDAL libraries are capable of working with most of them, a list of which can be found here [19].

4.2 Data Properties

4.2.1 Data Levels

In some instances one may want to perform their own processing of raw satellite data, but for our purposes here, we will just be concerned with working with data that has already been evaluated for quality and averaged into regular time allotments.

It is useful to be familiar with the different types of 'levels' that are available, so that you may select the proper data to begin working with [16]:

Level	Code	Description
Level 0	L0	Raw instrument data
Level 1A	L1A	Reconstructed raw instrument data
Level 1B	L1B	Products containing geolocated and calibrated spectral radiance and solar irradiance data
Level 2	L2	Products derived from the L1B product. One file per orbit
Level 3	L3	Averaged global gridded products, screened for bad data points
Level 4	L4	Model output; derived variables

With this information, we can see that we are most interested in the Level 3 (L3) data sets.

4.3 Focus Datasets

The datasets that will be covered are those for which methods have been written in `Geosetup`.

For data that is accessible via an FTP server, any standard FTP-client can be used for downloading the data (e.g. Filezilla) [6]. On UNIX-like systems this can more easily be done by using the *wget* command.

4.3.1 Pathfinder

Pathfinder is a merged data product from the National Oceanographic Data Center (NODC) [23], an office of the National Oceanic and Atmospheric Association (NOAA).

Summary

- *Data Format:* HDF5
- *Date Range:* 1981-2011 (gap from 1994-05-27 to 1995-07-01)
- *Timestamps:* in days from reference time '1981-01-01 00:00:00'
- *Data quality control:* variable containing quality flags for data points

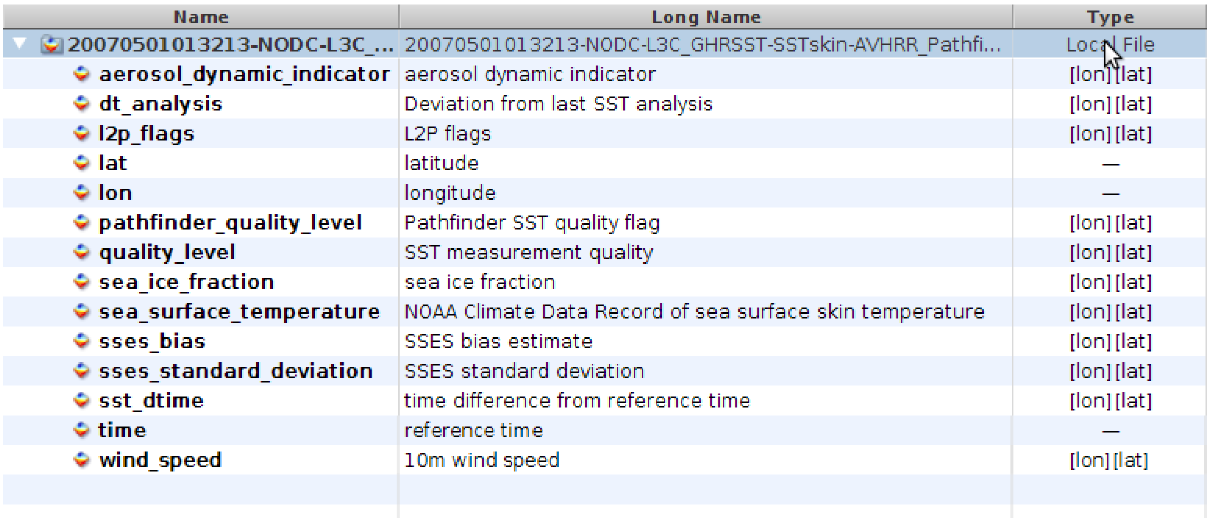
Variables in Dataset

Downloading

Pathfinder data is available to download via HTTP, FTP, THREDDS (supporting OPeNDAP), and rendered images/KML [23].

Example for downloading all Pathfinder v5.2 data for 2007:

```
wget -r -np -nH ftp://ftp.nodc.noaa.gov/pub/data.nodc/pathfinder/Version5.2/2007
```



Name	Long Name	Type
20070501013213-NODC-L3C_...	20070501013213-NODC-L3C_GHRSST-SSTskin-AVHRR_Pathfi...	Local File
aerosol_dynamic_indicator	aerosol dynamic indicator	[lon] [lat]
dt_analysis	Deviation from last SST analysis	[lon] [lat]
l2p_flags	L2P flags	[lon] [lat]
lat	latitude	—
lon	longitude	—
pathfinder_quality_level	Pathfinder SST quality flag	[lon] [lat]
quality_level	SST measurement quality	[lon] [lat]
sea_ice_fraction	sea ice fraction	[lon] [lat]
sea_surface_temperature	NOAA Climate Data Record of sea surface skin temperature	[lon] [lat]
sses_bias	SSES bias estimate	[lon] [lat]
sses_standard_deviation	SSES standard deviation	[lon] [lat]
sst_dtime	time difference from reference time	[lon] [lat]
time	reference time	—
wind_speed	10m wind speed	[lon] [lat]

Figure 4.1: Panoply output showing datasets included in Pathfinder v5.2 datafile

4.3.2 CoRTAD

CoRTAD is a reprocessed weekly average dataset of the Pathfinder data, developed by National Oceanographic Data Center (NODC) [22] for modelling and management of coral reef systems.

Summary

- *Data Format:* HDF5
- *Date Range:* 1981-2011 (gap from 1994-05-27 to 1995-07-01)
- *Timestamps:* in days from reference time '1981-01-01 00:00:00'
- *Data quality control:* Two separate data variables are included, including one where gaps due to clouds, etc. have been interpolated. Also included is an array of all bad data, and a variety of variables providing statistical evaluation of all data.

Variables in Dataset

Downloading

On UNIX-like systems, you may use a file with a list of the URL's for all files you would like to download. Then you can use the `wget` command with the `-i` option to download all files in the list.

Example cortad_wget-list.txt

```
ftp://ftp.nodc.noaa.gov/pub/data.nodc/cortad/Version4/cortadv4_row01_col05.nc
ftp://ftp.nodc.noaa.gov/pub/data.nodc/cortad/Version4/cortadv4_row00_col09.nc
ftp://ftp.nodc.noaa.gov/pub/data.nodc/cortad/Version4/cortadv4_row01_col09.nc
```

Example wget command:

Name	Long Name	Type
cortadv4_row00_col05.nc	cortadv4_row00_col05.nc	Local File
AllBad	A matrix showing which pixels are always missing. These mi...	[lon] [lat]
ClimSST	Climatological SST based on weekly SSTs for 1981-2010, cr...	[lon] [lat]
crs	crs	—
FilledSST	Gap-free weekly SST time series for 1982-2009 after media...	[lon] [lat] [time]
Harmonics	The five coefficients determined using the fit harmonics ap...	[lon] [lat]
Land	A matrix showing which pixels are on land. No analyses take...	[lon] [lat]
lat	latitude	—
lat_bounds	lat_bounds	—
lon	longitude	—
lon_bounds	lon_bounds	—
MedfillSST	Partially gap-filled weekly SST time series after median fill fo...	[lon] [lat] [time]
NumberGood	A matrix showing how many of the original SST observations...	[lon] [lat]
SST_Stats	Minimum, Maximum, Standard Deviation, and Mean of SST ti...	[lon] [lat]
SSTA	Weekly SST Anomalies	[lon] [lat] [time]
SSTA_DHW	Sum of SST Anomalies >= 1 deg C over previous 12 weeks (...	[lon] [lat] [time]
SSTA_DHW_Stats	Maximum, Standard Deviation, and Mean of SST Anomaly De...	[lon] [lat]
SSTA_Frequency	Frequency of SST Anomalies >= 1 deg C over previous 52 w...	[lon] [lat] [time]
SSTA_Frequency_Stats	Maximum, Standard Deviation, and Mean of SST Anomaly Fr...	[lon] [lat]
SSTA_Stats	Minimum, Maximum, Standard Deviation, Mean, and Mean of...	[lon] [lat]
time	reference time	—
time_bounds	time_bounds	—
TSA	Thermal Stress Anomalies (defined as SST minus Maximum ...	[lon] [lat] [time]
TSA_DHW	Sum of SST Anomalies >= 1 deg C over previous 12 weeks (...	[lon] [lat] [time]
TSA_DHW_Stats	Maximum, Standard Deviation, and Mean of Thermal Stress ...	[lon] [lat]
TSA_Frequency	Frequency of Thermal Stress Anomalies >= 1 deg C over pr...	[lon] [lat] [time]
TSA_Frequency_Stats	Maximum, Standard Deviation, and Mean of Thermal Stress ...	[lon] [lat]
TSA_Stats	Maximum, Standard Deviation, and Mean of Thermal Stress ...	[lon] [lat]
WeeklySST	Gappy weekly SST time series for 1981-2010	[lon] [lat] [time]

Figure 4.2: Panoply output showing datasets included in CoRTAD datafile

```
wget -i cortad_wget-list.txt
```

4.3.3 GlobColour Chlorophyll-a Data

The European Space Agency (ESA) produces a merged chlorophyll dataset that combines data acquired from the SeaWiFS (NASA), MODIS (NASA), and MERIS (ESA) satellite imaging missions.

A very extensive explanation of the data and its derivation is covered in the GlobColour Product User Guide. You may download it from their website here:

http://www.globcolour.info/CDR_Docs/GlobCOLOUR_PUG.pdf

This merged data product is ideal for applications where your sampling period may extend beyond that covered by any one when your sampling period extends across the acquisition periods of each of the different missions offering imagery in a spectral range from which chlorophyll values may be processed from.

Note: See *Temporal ranges of ocean color satellite missions* for coverage of different datasets.

Different merging methods have been used, including:

- simple averaging
- weighted averaging
- GSM model

Grid Formats

Integerized Sinusoidal (ISIN) L3b Product

- Angular resolution: $1/24°$ (approx. $4.63km$ spatial resolution)
- Uses `rows` and `cols` arrays to specify the latitudinal and longitudinal index of the bins, stored in the product (i.e. mean chlorophyll values in `CHL1_mean`)

Plate-Carré projection (Mapped) L3m Product

- Angular resolution: $0.25°$ and $1.0°$
- latitude and longitude specifying the center of each cell

Note: The mapped (L3m) product is an easier format to work, as it uses standard coordinate and value organization, but you will need to use the ISIN grid (L3b) product if you require th ~4.63 km resolution.

Summary

- *Data Format:* netCDF-3
- *Date Range:* 1981-2011 (gap from 1994-05-27 to 1995-07-01)
- *Timestamps:* in hours from reference time '1981-01-01 00:00:00'
- *Data quality control:* variable containing quality flags for data points

Variables in Dataset

Name	Long Name	Type
L3m_20070501__GLOB_25_GS...	L3m_20070501__GLOB_25_GSM-MERMODSWF_CHL1_DAY_00.nc	Local File
CHL1_error	Chlorophyll-a concentration. Case 1 water - Error estimation	[lon] [lat]
CHL1_flags	Chlorophyll-a concentration. Case 1 water - Flags	[lon] [lat]
CHL1_mean	Chlorophyll-a concentration. Case 1 water - Mean of the bin...	[lon] [lat]
lat	latitude	—
lon	longitude	—

Figure 4.3: Panoply output showing datasets included in Globcolour datafile

Downloading

Data can be downloaded for specific time periods and geographic extents through their gui download interface on the web:

http://hermes.acri.fr/GlobColour/index.php

ESA's also maintains an ftp-server from which you may download GlobColour data in batch, for a large number of files.

Example for downloading GSM averaged 1-Day Chlorphyll-a between 1997-2012:

```
wget -r -l10 -t10 -A "*.gz" -w3 -Q1000m \
ftp://globcolour_data:fgh678@ftp.acri.fr/GLOB_4KM/RAN/CHL1/MERGED/GSM/{1997,1998
```

4.3.4 GEBCO Bathymetric Data

The General Bathymetric Chart of the Oceans is a compilation of a ship depth soundings and satellite gravity data and images. Using quality-controlled data to begin with, values between sounded depths have been interpolated with gravity data obtained by satellites [13].

Source Identifier (SID) Grid describes which measurements are from soundings or predictions.

Summary

- *Data Format:* netCDF-3
- ***Angular resolution:*** $30sec$ **and** 1° **grids**

  - `(0,0)` position at northwest corner of file
  - Grid pixels are center-registered
- ***Data organization:*** **Single data file for whole globe**
  - 21,600 rows x 43,200 columns = 933,120,000 data points
- *Units:* depth in meters. Bathymetric depths are negative and topographic positive
- *Date Range:* Dates of soundings vary. Most recent 2010
- *Data quality control:* separate file with SID grid of data sources and quality.

Variables in Dataset

Name	Long Name	Type
gebco_08.nc	gebco_08.nc	Local File
dimension	dimension	—
spacing	spacing	—
x_range	x_range	—
y_range	y_range	—
z	z	—
z_range	z_range	—

Figure 4.4: Panoply output showing datasets included in GEBCO datafile

Downloading

Before downloading the GEBCO data, you mush register at their website first. Follow link to register

After registration, you can access the data here. You will have the option of downloading either the `gebco_08.nc` file for the 30sec gridded data or the `gridone.nc` file for the 1°.

You may also download the SID grid which contains flags corresponding to each data point as to whether it is a sounding or interpolated, and the quality of each.

4.4 Previewing Data

Various applications are available for viewing netCDF and HDF formatted data, but not all are capable of opening versions of both formats such as both HDF5 and netCDF3.

When writing code to work with such data sets, it is often necessary to know the names and dimensions of variables withing the datafile, which a viewing application makes particularly easy.

4.4.1 Panoply

Panoply is a viewer application developed by NASA that allows a number of useful tools for exploring data in netCDF, HDF, and GRIB formats.

Note: Panoply is dependent on the Java runtime environment, which must be installed before installing. Download Java

---

Panoply Download Link:

http://www.giss.nasa.gov/tools/panoply/

Some Neat Features

- Slice and plot data sets, combining them, and save to image files
- Connect to OPeNDAP and THREDDS data servers to explore data remotely
- Create animations from collection of images

4.4.2 VISAT Beam

The ESA had commissioned the development BEAM of an open-source set of tools for viewing, analyzing and processing are large number of different satellite data products [14]. In addition they have developed the GUI desktop application VISAT for using these tools

Dowloand LSAT/BEAM

4.4.3 Quantum GIS

Quantum GIS is an open source GIS application that is available for all major operating systems and has features similar to those found in much more costly GIS applications.

It is an excellent resource when viewing images created by your scripts, and has the ability to be extended with PyQGIS once you become familiar with Python and working with geospatial data.

Download Quantum GIS

CHAPTER FIVE

GETTING STARTED WITH PYTHON

Quickstart

Skip to installation and package guide

If you have previous experience programming, learning Python is a fairly intuitive process with syntax that is easily read and remembered compared to other lower level languages like C/C++.

A good place to start is with Google's free online python course:

> https://developers.google.com/edu/python/

5.1 Science and Python

Along with gaining a general familiarity in Python, you will want to become familiar with the use of NumPy, as this module is used extensively for working with large datasets in an efficient manner. In addition the `netCDF4` module that is used here for opening `netCDF` and `HDF` datasets by default creates masked `NumPy` array when importing these datasets.

The SciPy team has a great NumPy tutorial to get started with:

> http://www.scipy.org/Tentative_NumPy_Tutorial

Though not really covered in this article, the Python Data Analysis Library (Pandas) has incredible tools for data processing and working with statistical analysis and modelling. In combination together, Python, NumPy/SciPy, and Pandas offers a powerful data processing ability, rivalling `R` and `Matlab` with functionality that does not exist in either .

5.2 Troubleshooting

iPython is an excellent tool for working interactively with python, allowing you to test code, play with concepts you are exploring, and store various information in a workspace similar to `R` and `Matlab` work flows [2] [21].

When hitting stumbling blocks or just trying to improve your code, StackOverflow is a fantastic resource for receiving support and finding existing solutions to your problems.

5.3 Important things to consider

- Backup! In more than two places preferably. It takes a long time to code, and you don't want to start over if something happens.
- Along with general backups, using a version control system such as Git, Subversion, or Mercurial. Doing so allows you to easily revert to previous versions of your code, collaborate with others, track problems, develop in stages, and more.
- Stick to the conventions: coding like everybody else (e.g. naming, spacing, commenting, etc.) makes things easier for you and for others to work with your code.

5.4 Setting up your development environment

The code here was developed using Python version 2.7; however it should be compatible with versions greater the 2.6.

5.4.1 Installing Python

Python can be downloaded from Python.org as source, as well as binary installers for Windows and Mac OSX.

Most distributions of Linux come with Python pre-installed. To see what version of python you are running, run the following from the command line `python -V`.

5.4.2 Additional Python Packages you will need

In order to properly run the code presented here, you will need to also install a number of other libraries/packages that are used.

It will just be assumed that if you are using another distribution of Linux/UNIX than Ubuntu, that you will be familiar enough to hunt down the packages/source and install them.

- **gdal**
  - Ubuntu (linux): https://launchpad.net/~ubuntugis/+archive/ppa/
  - Windows: Install OSGeo (with GDAL): http://trac.osgeo.org/osgeo4w/wiki
  - Mac OSX: Pre-compiled binaries http://www.kyngchaos.com/software/frameworks#gdal_com
- **pyroj**
  - Ubuntu (linux): http://packages.ubuntu.com/search?keywords=python-pyproj
  - Windows: http://code.google.com/p/pyproj/downloads/list
  - Mac OSX (Unverified): http://mac.softpedia.com/get/Math-Scientific/pyproj.shtml

- **numpy**
  - Ubuntu (linux): from command line `sudo apt-get install python-numpy`
  - Windows: http://sourceforge.net/projects/numpy/files/NumPy/
  - Mac OSX: http://www.scipy.org/Installing_SciPy/Mac_OS_X
- **scipy**
  - Ubuntu (linux): from command line `sudo apt-get install python-scipy`
  - Windows: http://www.scipy.org/Installing_SciPy/Windows
  - Mac OSX: http://www.scipy.org/Installing_SciPy/Mac_OS_X
- **matplotlib**
  - All operating systems: http://matplotlib.org/faq/installing_faq.html#installation
- **matplotlib.basemap**
  - To be installed as above, just note that it must be installed in addition to matplotlib

5.4.3 Getting Geosetup

In order to collaborate to the code, it is preferable that you obtain the `Geosetup` code by forking the repository from GitHub.

If you're new to Git, a comprehensive instructional book is available here [3]:

If you prefer you may download the code as a ZIP archive, or access it from the repository page on GitHub.

CHAPTER

SIX

IMPORTING NETCDF AND HDF DATA WITH PYTHON

The first thing you'll need to do to work with these datasets is to import it to

6.1 netCDF4

netCDF4 is a python package that utilizes `NumPy` to read and write files in netCDF and HDF formats [12]. It contains methods that allow the opening and writing of `netCDF4`, `netCDF3` and `HDF5` files.

Before importing the data, you'll want to find the name of the dataset variable that you want to import. You can determine this by first opening the file with *Panoply* and checking the name of the variable. Examples for the different variables available from this paper's focus datasets is found in the summaries sections of *Focus Datasets*.

Example showing the data import using netCDF4:

```
import numpy as np
import netCDF4 as Dataset

filepath = /path/to/file.nc
# Read in dataset (using 'r' switch, 'w' for write)
dataset = Dataset(filepath,'r')
# Copy data to variables, '[:]' also copies missing values mask
lons = dataset.variables["lon"][:]
lats = dataset.variables["lat"][:]
sst = dataset.variables["sea_surface_temperature][:]
dadtaset.close()
```

When data is copied to a variable in Python, it places it all into the systems temporary memory, which means there is a limit as to how much data can be loaded. With certain datasets, the dataset variables may be large enough to exceed the system memory.

Typically these datasets cover much larger geographic areas than are interested in (i.e. the whole earth), so it is possible to load only the data for the area of interest.

```
def getnetcdfdata(data_dir, nc_var_name, min_lon, max_lon, min_lat, max_lat, data_time_start, data_time_e
'''Extracts subset of data from netCDF file

based on lat/lon bounds and dates in the iso format '2012-01-31' '''

# Create list of data files to process given date-range
```

```
    file_list = np.asarray(os.listdir(data_dir))
    file_dates = np.asarray([datetime.datetime.strptime(re.split('-', filename)[0], '%Y%m%d%H%M%S') for filen
    data_files = np.sort(file_list[(file_dates >= data_time_start)&(file_dates <= data_time_end)])

    # Get Lat/Lon from first data file in list
    dataset = Dataset(os.path.join(data_dir,file_list[0]),'r')
    lons = dataset.variables["lon"][:]
    lats = dataset.variables["lat"][:]

    # Create indexes where lat/lons are between bounds
    lons_idx = np.where((lons > math.floor(min_lon))&(lons < math.ceil(max_lon)))[0]
    lats_idx = np.where((lats > math.floor(min_lat))&(lats < math.ceil(max_lat)))[0]
    x_min = lons_idx.min()
    x_max = lons_idx.max()
    y_min = lats_idx.min()
    y_max = lats_idx.max()

    # Create arrays for performing averaging of files
    vals_sum = np.zeros((y_max-y_min + 1, x_max-x_min + 1))
    vals_sum = ma.masked_where(vals_sum < 0 , vals_sum)
    mask_sum = np.empty((y_max-y_min + 1, x_max-x_min + 1))
    dataset.close()

        # Get average of chlorophyl values from date range
        file_count = 0
        for data_file in data_files:
            current_file = os.path.join(data_dir,data_file)
            dataset = Dataset(current_file,'r') # by default numpy masked array
            vals = np.copy(dataset.variables[nc_var_name][0 ,y_min:y_max + 1, x_min:x_max + 1])
            #vals = ma.masked_where(vals < 0, vals)
            #ma.set_fill_value(vals, -999)
            vals = vals.clip(0)
            vals_sum += vals
            file_count += 1
            dataset.close()
        vals_mean = vals_sum/file_count #TODO simple average

        # Mesh Lat/Lon the unravel to return lists
        lons_mesh, lats_mesh = np.meshgrid(lons[lons_idx],lats[lats_idx])
        lons = np.ravel(lons_mesh)
        lats = np.ravel(lats_mesh)
        vals_mean = np.ravel(vals_mean)

        return lons, lats, vals_mean
```

Another option for avoiding this problem is by loading the data in chunks, using sparse arrays, or python libraries that allow loading the data onto the system hard drive, such as PyTables or h5py.

6.1.1 OPeNDAP & THREDDS

Rather than opening the data from files on your local machine, it is possible to open them remotely. Rather than creating the dataset from a local data file path, you simply reference the path to a file on a remote `OPeNDAP` or `THREDDS` server.

To create a dataset for data located at `http://data.nodc.noaa.gov/opendap/pathfinder/Vers` you would simply do the following:

```
import numpy as np
import netCDF4 as Dataset

filepath = 'http://data.nodc.noaa.gov/opendap/pathfinder/Version5.2/1981/19811101025755-NODC-L3C_GHRSST-S
# Read in dataset (using 'r' switch, 'w' for write)
dataset = Dataset(filepath,'r')
```

For further information on these protocols, thorough explanations are presented on their websites.

OPeNDAP

THREDDS

CHAPTER

SEVEN

WORKING WITH PROJECTIONS USING PYTHON

7.1 pyproj

pyproj is a python interface to the PROJ.4 library which offers methods for working between geographic coordinates (e.g. latitude and longitude) and Cartesian coordinates (i.e. two dimensional projected coordinates - x,y) [4].

7.1.1 Defining variables from pyproj class instances

It is possible to extract information from pyproj ellipsoid and projection class instances if needed for other calculations.

As an example, let us first define an ellipsoid using the WGS84 datum:

```
g = pyproj.Geod(ellps='WGS84')
```

If we were to want to use the radius of the earth of this ellipsoid for another calculation, we could extract the value to a variable from the pyproj ellipsoid class instance we defined above.

To find the value, you can use the `dir()` function to find the names that are defined by a module. We can see by the following that the first variable listed (i.e. not an `__attribute__`) is `a`:

```
dir(g)
>>> dir(g)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
```

You may access these by using either `getattr` function (if you have the property name as a string or would like to access it dynamically) or more simply when we know the name of the property by just using the dot notation:

```
>>> getattr(g,'a') # getattr function
6378137.0

>>> g.a # dot notation
6378137.0
```

If we hadn't already known that a was the value we were looking for, we could have just used the dot notation as above to have output the various values that were available in this object.

To make things easier to read later, you could do the following:

```
earth_equa_radius = g.a # earth's radius at the equator in meters
earth_pole_radius = g.b # earth's radius through poles in meters
```

7.1.2 Defining a projection

Example of defining a projection using the EPSG code:

```
import pyproj

# LatLon with WGS84 datum used by GPS units and Google Earth
wgs84 = pyproj.Proj(init='EPSG:4326')

# Lambert Conformal Conical (LCC)
lcc = pyproj.Proj(init='EPSG:3034')
```

Not all projections are available by using the ESPG reference code, but it is possible to define your own projection using pyproj and gdal in python.

Spatialreference is a great resource for finding the details for the projection you'd like to create (assuming it's just not in the library; though, feel free to define a completely custom projection).

Example of defining the Albers projection:

```
# Albers Equal Area Conic (aea)
nplaea1 = pyproj.Proj("+proj=laea +lat_0=90 +lon_0=-40 +x_0=0 +y_0=0 \
                   +ellps=WGS84 +datum=WGS84 +units=m +no_defs")
```

CHAPTER

EIGHT

GRIDDING AND RESAMPLING DATA WITH PYTHON

When using remote sensing data for model applications, you will want to somehow standardize everything to a uniform grid. Your data may be scattered, and the satellite data be arranged at different resolutions and grids (and perhaps using different projections).

8.1 Deciding what to do

8.1.1 Choosing an Interpolation Method

Depending on the type of data that you will be re-sampling to a grid, different interpolation methods may be better suited to your data.

It is possible to interpolate data referenced by an unprojected coordinate system; though, due to more complicated calculations involved and potential differences in the datasets, it may be easiest to first project the data. The type of projection that is chosen to project the data to before interpolating it using a 2-dimensional interpolation method depends upon the exent of the study area and latitudes at which the data exist.

8.1.2 Choosing a Cartesian Projection

In general, you should attempt to choose a projection that does not distort the area over which the area exists, introducing error in you interpolation product, such as using the Albers Equal Area projection for a large area extending into northern latitudes (e.g. the North Atlantic ocean) (see *Map Projections*).

8.2 Gridding

Before interpolating the data to a new grid, you will want to create that grid, defined by latitude and longitude coordinates.

8.2.1 Creating a grid

Create Grid:

```
import numpy as np

def creategrid(min_lon, max_lon, min_lat, max_lat, cell_size_deg, mesh=False):
    '''Output grid within geobounds and specifice cell size

       cell_size_deg should be in decimal degrees'''

    min_lon = math.floor(min_lon)
    max_lon = math.ceil(max_lon)
    min_lat = math.floor(min_lat)
    max_lat = math.ceil(max_lat)

    lon_num = (max_lon - min_lon)/cell_size_deg
    lat_num = (max_lat - min_lat)/cell_size_deg

    grid_lons = np.zeros(lon_num) # fill with lon_min
    grid_lats = np.zeros(lat_num) # fill with lon_max
    grid_lons = grid_lons + (np.assary(range(lon_num))*cell_size_deg)
    grid_lats = grid_lats + (np.assary(range(lat_num))*cell_size_deg)

    grid_lons, grid_lats = np.meshgrid(grid_lons, grid_lats)
    grid_lons = np.ravel(grid_lons)
    grid_lats = np.ravel(grid_lats)

    #if mesh = True:
    #    grid_lons = grid_lons
    #    grid_lats = grid_lats

    return grid_lons, grid_lats
```

8.3 Re-sampling to Grid

8.3.1 Without Projecting

It is possible to take the geographic coordinates and interpolate them over a spherical surface, rather than first projecting the data, and then performing the interpolation on a 2-dimensional surface. This can be done using *scipy.interpolator.RectSphereBivariateSpline*'s smoothing spline approximation:

```
import numpy as np
from scipy.interpolate import RectSphereBivariateSpline

def geointerp(lats,lons,data,grid_size_deg, mesh=False):
    '''We want to interpolate it to a global x-degree grid'''
    deg2rad = np.pi/180.
    new_lats = np.linspace(grid_size_deg, 180, 180/grid_size_deg)
    new_lons = np.linspace(grid_size_deg, 360, 360/grid_size_deg)
    new_lats_mesh, new_lons_mesh = np.meshgrid(new_lats*deg2rad, new_lons*deg2rad)

    '''We need to set up the interpolator object'''
    lut = RectSphereBivariateSpline(lats*deg2rad, lons*deg2rad, data)

    '''Finally we interpolate the data. The RectSphereBivariateSpline
    object only takes 1-D arrays as input, therefore we need to do some reshaping.'''
    new_lats = new_lats_mesh.ravel()
    new_lons = new_lons_mesh.ravel()
    data_interp = lut.ev(new_lats,new_lons)

    if mesh == True:
```

```
        data_interp = data_interp.reshape((360/grid_size_deg,
                                           180/grid_size_deg)).T

    return new_lats/deg2rad, new_lons/deg2rad, data_interp
```

Limitations:

- This method is restricted to using the smoothing spline approximation, which may not be the best method for your data
- It assumes a spherical earth, where using the WGS84 datum would more accurately represent the data on the ellipsoid earth.

CHAPTER

NINE

PLOTTING WITH PYTHON

9.1 Matplotlib and Basemap

Matplotlib is a versatile plotting library that was developed by John Hunter that may be used for everything from basic scatter plots to cartographic plots, along with remote sensing data.

Matplotlib basemap is an additional library to `matplotlib` which may be used for plotting maps and 2-dimensional data on them. It performs similar functionality to `pyproj` (and uses the same `PROJ.4` libraries) to transform data to 2-dimensional projections.

9.1.1 Introduction

A collection of `matplotlib.basemap` example plotting scripts can be found here.

Just to get an idea, let's say we want to plot a map of Iceland in the Albers Conic Conformal projection

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

mapWidth = 600000
mapHeight = 600000
lat0center = 64.75
lon0center = -18.60

m = Basemap(width=mapWidth,height=mapHeight,
        rsphere=(6378137.00,6356752.3142),\
        resolution='f',area_thresh=1000.,projection='lcc',\
        lat_1=45.,lat_2=55.,\
        lat_0=lat0center,lon_0=lon0center)

# Draw parallels and meridians.
m.drawparallels(np.arange(-80.,81.,1.), labels=[1,0,0,0], fontsize=10)
m.drawmeridians(np.arange(-180.,181.,1.), labels=[0,0,0,1], fontsize=10)
m.drawmapboundary(fill_color='aqua')
m.drawcoastlines(linewidth=0.2)
m.fillcontinents(color='white', lake_color='aqua')
plt.show()
```

If you would like to plot data points on a map, and have the points show as increasingly larger markers, with increasing values, here is how you do it:

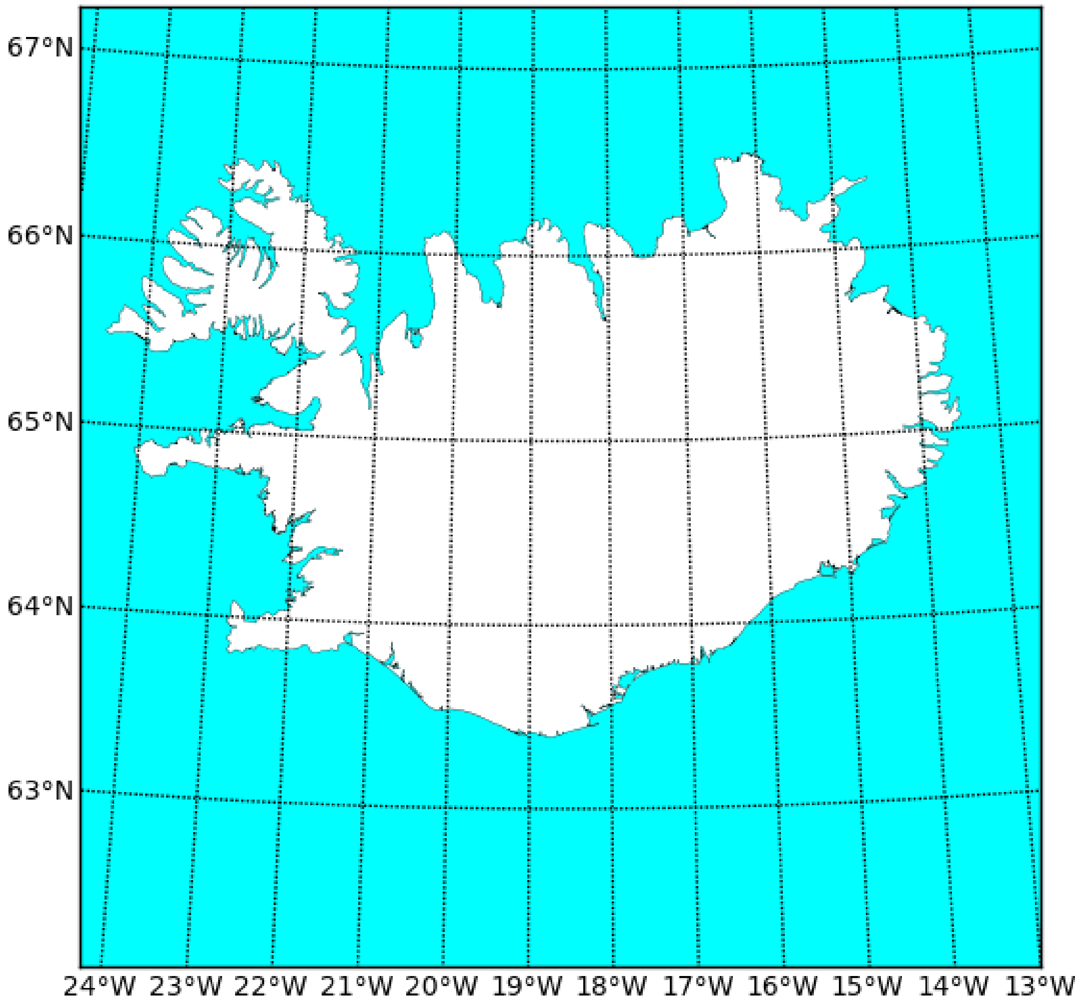


Figure 9.1: Plot of Iceland using matplotlib's basemap library with one degree parallels and meridians.

```
def plotSizedData(map_obj,lons,lats,values,symbol,min_size,max_size,lines=False):
    '''
    Plot data with varying sizes
    '''
    proj_x,proj_y = m(lons,lats)

    # Calculate marker sizes using y=mx+b
    # where y = marker size and x = data value
    slope = (max_size-min_size)/(max(values)-min(values))
    intercept = min_size-(slope*min(values))

    for x, y, val in zip(proj_x, proj_y, values):
        msize = (slope*val)+intercept
        map_obj.plot(x, y, symbol, markersize=msize)
```

To automatically center a map projected in a conic conformal projection, the following function will determine the correct center latitude, longitude, and proper width and height to encompass the data. You may also pass a scale to the function to make the width and height a desire percentage larger than the minimum plot area, so that there is a margin around the outermost points on the plot.

```
def centerMap(lons,lats,scale):
    '''
    Set range of map. Assumes -90 < Lat < 90 and -180 < Lon < 180, and
    latitude and logitude are in decimal degrees
    '''
    north_lat = max(lats)
    south_lat = min(lats)
    west_lon = max(lons)
    east_lon = min(lons)

    # find center of data
    # average between max and min longitude
    lon0 = ((west_lon-east_lon)/2.0)+east_lon

    # define ellipsoid object for distance measurements
    g = pyproj.Geod(ellps='WGS84') # Use WGS84 ellipsoid TODO make variable
    earth_radius = g.a # earth's radius in meters

    # Use pythagorean theorom to determine height of plot
    # divide b_dist by 2 to get width of triangle from center to edge of data area
    # inv returns [0]forward azimuth, [1]back azimuth, [2]distance between

    # a_dist = the height of the map (i.e. mapH)
    b_dist = g.inv(west_lon, north_lat, east_lon, north_lat)[2]/2
    c_dist = g.inv(west_lon, north_lat, lon0, south_lat)[2]

    mapH = pow(pow(c_dist,2)-pow(b_dist,2),1./2)
    lat0 = g.fwd(lon0,south_lat,0,mapH/2)[1]

    # distance between max E and W longitude at most southern latitude
    mapW = g.inv(west_lon, south_lat, east_lon, south_lat)[2]

    return lon0, lat0, mapW*scale, mapH*scale
```

CHAPTER

TEN

WRITING GEOSPATIAL DATA TO FILES

Say you want to take the data that you have been working on, maybe do some calculations on, and put it into a file that you can import into a GIS application such as ArcGIS or Quantum GIS. This could be done by writing the data to a text format, such as ESRI's ASCII grid format, but a more efficient way of doing this is by writing it to a GeoTIFF format using the bindings for GDAL libraries (TODO link).

10.1 Write file function

Imports:

```
import numpy as np
import gdal
import os
from osgeo import gdal
from osgeo import osr
from osgeo import ogr
from osgeo.gdalconst import *
import pyproj
import scipy.sparse
import scipy
gdal.AllRegister() #TODO remove? not necessary?
gdal.UseExceptions()
```

`GeoPoint` Class for transforming points and writing to raster files:

```
class GeoPoint:
    '''Tranform coordinate system, projection, and write geospatial data'''

    """lon/lat values in WGS84"""
    # LatLon with WGS84 datum used by GPS units and Google Earth
    wgs84 = pyproj.Proj(init='EPSG:4326')
    # Lambert Conformal Conical (LCC)
    lcc = pyproj.Proj(init='EPSG:3034')
    # Albers Equal Area Conic (aea)
    nplaea1 = pyproj.Proj("+proj=laea +lat_0=90 +lon_0=-40 +x_0=0 +y_0=0 \
                          +ellps=WGS84 +datum=WGS84 +units=m +no_defs")
    # North pole LAEA Atlantic
    nplaea2 = pyproj.Proj(init='EPSG:3574')
    # WGS84 Web Mercator (Auxillary Sphere; aka EPSG:900913)
    # Why not to use the following: http://gis.stackexchange.com/a/50791/12871
    web_mercator = pyproj.Proj(init='EPSG:3857')
    # TODO add useful projections, or hopefully make automatic

```

```
        def __init__(self, x, y, vals,inproj=wgs84, outproj=nplaea,
                                cell_width_meters=50.,cell_height_meters=50.):
            self.x = x
            self.y = y
            self.vals = vals
            self.inproj = inproj
            self.outproj = outproj
            self.cellw = cell_width_meters
            self.cellh = cell_height_meters

        def __del__(self):
            class_name = self.__class__.__name__
            print class_name, "destroyed"
    def transform_point(self, x=None, y=None):
            if x is None:
                x = self.x
            if y is None:
                y = self.y
            return pyproj.transform(self.inproj,self.outproj,x,y)


        def get_raster_size(self, point_x, point_y, cell_width_meters, cell_height_meters):
            """Determine the number of rows/columns given the bounds of the point
            data and the desired cell size"""
            # TODO convert points to 2D projection first

            cols = int(((max(point_x) - min(point_x)) / cell_width_meters)+1)
            rows = int(((max(point_y) - min(point_y)) / abs(cell_height_meters))+1)

            print 'cols: ',cols
            print 'rows: ',rows

            return cols, rows


        def create_geotransform(self,x_rotation=0,y_rotation=0):
            '''Create geotransformation for converting 2D projected point from
            pixels and inverse geotransformation to pixels (what I want, but need the
            geotransform first).'''

            # TODO make sure this works with negative & positive lons
            top_left_x, top_left_y = self.transform_point(min(self.x),
                                                          max(self.y))

            lower_right_x, lower_right_y = self.transform_point(max(self.x),
    # GeoTransform parameters
            # --> need to know the area that will be covered to define the geo tranform
            # top left x, w-e pixel resolution, rotation, top left y, rotation, n-s pixel resolution
            # NOTE: cell height must be negative (-) to apply image space to map
            geotransform = [top_left_x, self.cellh, x_rotation,
                             top_left_y, y_rotation, -self.cellh]

            # for mapping lat/lon to pixel
            success, inverse_geotransform = gdal.InvGeoTransform(geotransform)
            if not success:
                print 'gdal.InvGeoTransform(geotransform) failed!'
                sys.exit(1)

            return geotransform, inverse_geotransform
```

Convert coordinate to pixel:

```
    def point_to_pixel(self, point_x, point_y, inverse_geotransform):
            """Translates points from input projection (using the inverse
            transformation of the output projection) to the grid pixel coordinates in data
            array (zero start)"""

            # apply inverse geotranformation to convert to pixels
            pixel_x, pixel_y = gdal.ApplyGeoTransform(inverse_geotransform,
                                                      point_x, point_y)

            return pixel_x, pixel_y
```

Convert pixel to coordinate:

```
     def pixel_to_point(self, pixel_x, pixel_y, geotransform):
            """Translates grid pixels coordinates to output projection points
            (using the geotransformation of the output projection)"""

            point_x, point_y = gdal.ApplyGeoTransform(geotransform, pixel_x, pixel_y)
    return point_x, point_y
```

Write data to raster file:

```
     def create_raster(self, in_x=None, in_y=None, filename="data2raster.tiff",
    output_format="GTiff", cell_width_meters=1000., cell_height_meters=1000.):
            '''Create raster image of data using gdal bindings'''
            # if coords not provided, use default values from object
            if in_x is None:
                in_x = self.x
            if in_y is None:
                in_y = self.y

            # create empty raster
            current_dir = os.getcwd()+'/'
            driver = gdal.GetDriverByName(output_format)
            number_of_bands = 1
            band_type = gdal.GDT_Float32
            NULL_VALUE = 0
            self.cellw = cell_width_meters
            self.cellh = cell_height_meters

            geotransform, inverse_geotransform = self.create_geotransform()

            # convert points to projected format for inverse geotransform
            # conversion to pixels
            points_x, points_y = self.transform_point(in_x,in_y)
            cols, rows = self.get_raster_size(points_x, points_y, cell_width_meters, cell_height_meters)
            pixel_x = list()
            pixel_y = list()
            for point_x, point_y in zip(points_x,points_y):
                # apply value to array
                x, y = self.point_to_pixel(point_x, point_y, inverse_geotransform)
                pixel_x.append(x)
                pixel_y.append(y)

            dataset = driver.Create(current_dir+filename, cols, rows, number_of_bands, band_type)
    # Set geographic coordinate system to handle lat/lon
            srs = osr.SpatialReference()
            srs.SetWellKnownGeogCS("WGS84") #TODO make this a variable/generalized
            dataset.SetGeoTransform(geotransform)
            dataset.SetProjection(srs.ExportToWkt())

            #NOTE Reverse order of point components for the sparse matrix
            data = scipy.sparse.csr_matrix((self.vals,(pixel_y,pixel_x)),dtype=float)

            # get the empty raster data array
            band = dataset.GetRasterBand(1) # 1 == band index value

            # iterate data writing for each row in data sparse array
            offset = 1 # i.e. the number of rows of data array to write with each iteration
            for i in range(data.shape[0]):
                data_row = data[i,:].todense() # output row of sparse array as standard array
                band.WriteArray(data_row,0,offset*i)
            band.SetNoDataValue(NULL_VALUE)
            band.FlushCache()

            # set dataset to None to "close" file
            dataset = None
```

Main Program:

```python
if __name__ == '__main__':

    # example coordinates, with function test
    lat = [45.3,50.2,47.4,80.1]
    lon = [134.6,136.2,136.9,0.5]
    val = [3,6,2,8]

    # TODO clean-up test
    # Generate some random lats and lons
    #import random
    #lats = [random.uniform(45,75) for r in xrange(500)]
    #lons = [random.uniform(-2,65) for r in xrange(500)]
    #vals = [random.uniform(1,15) for r in xrange(500)]

    lats = np.random.uniform(45,75,500)
    lons = np.random.uniform(-2,65,500)
    vals = np.random.uniform(1,25,500)
    print 'lats ',lats.shape
    #test data interpolation
    import datainterp
    lats, lons, vals = datainterp.geointerp(lats,lons,vals,2,mesh=False)
    print 'lons ',lons.shape
    geo_obj =  GeoPoint(x=lons,y=lats,vals=vals)
    geo_obj.create_raster()
```

CHAPTER

ELEVEN

GEOSETUP - TEXTDATA

This module can be modified to any comma separated value or text file data set That you would like to import alongside remote sensing data you have imported. Currently it is configured to import a dataset of sight surveying data, assigning data types and sizes to the various columns in the dataset.

Imports:

```
import numpy as np
from netCDF4 import Dataset
import sys, os
from StringIO import StringIO
import datetime
```

Get data:

```
def getData(data_file, skip_rows=0):

    with open(data_file) as fh:
        file_io = StringIO(fh.read().replace(',', '\t'))

    # Define names and data types for sighting data
    record_types = np.dtype([
                    ('vessel',str,1),           #00 - Vessel ID
                    ('dates',str,6),            #01 - Date
                    ('times',str,6),            #02 - Time (local?)
                    ('lat',float),              #03 - latitude dec
                    ('lon',float),              #04 - longitude dec -1
                    ('beafort',str,2),          #05 - beafort scale
                    ('weather',int),            #06 - weather code
                    ('visibility',int),         #07 - visibility code
                    ('effort_sec',float),       #08 - seconds on effort
                    ('effort_nmil',float),      #09 - n miles on effort
                    ('lat0',float),             #10 - lat of sight start
                    ('lon0',float),             #11 - lon of sight start
                    ('num_observers',int),      #12 - number of observers
                    ('species',str,6),          #13 - species codes
                    ('num_animals',int),        #14 - number observed
                    ('sighting',int),           #15 - Boolean sight code
                    ('rdist',float),            #16 - distance to sight
                    ('angle',float),            #17 - angel from ship
                    ('block',str,2),            #18 - cruise block
                    ('leg',int),                #19 - cruise leg code
                    ('observations',str,25),    #20 - cruise obs codes
                    ])

    # Import data to structured array
    data = np.genfromtxt(file_io, dtype = record_types, delimiter = '\t',
                         skip_header = skip_rows)

    # Correct longitude values
    data['lon'] = data['lon']*(-1)
```

```python

    # Print a summary of geo data
    print '\nSighting Data Information'
    print '-------------------------------------------'
    print 'Data Path: ' + data_file
    print 'First sighting: ', min(data['dates'])
    print 'Last sighting: ', max(data['dates'])

    return data

    #TODO remove following
    # TODO calculate distance between start points and sighting point and
    # compare

    # Create array of unique survey block IDs
    #blocks = np.unique(data['block'])

    #g = pyproj.Geod(ellps='WGS84') # Use WGS84 ellipsoid
    #f_azimuth, b_azimuth, dist =
    #g.inv(data['lon'],data['lat'],data['lon0'],data['lat0'])

    #last_idx = 0
    #idx_pos = 0
    #minke_effort = np.zeros_like(minke_idx, dtype=float)
    #for idx in minke_idx:
    #    minke_effort[idx_pos] = dist[last_idx:(idx+1)].sum()
    #    last_idx = idx
    #    idx_pos = idx_pos+1

    # generate list of indexes where effort was greater than zero
    #effort_idx = np.where(data['effort_sec']*data['effort_nmil'] != 0)

    # spue = data['num_animals'][effort_idx]/(data['effort_sec'][effort_idx]*data['effort_nmil'][effort_

    # append spue calculations to structured array dataset
    #data = np.lib.recfunctions.append_fields(data,'spue',data=spue)
```

Main program:

```python
if __name__ == '__main__':

    ####################
    # commandline usage #
    ####################
    if len(sys.argv) < 2:
        print >>sys.stderr,'Usage:',sys.argv[0],'<data directory>\n'
        sys.exit(1)

    data_file = sys.argv[1]

    data, geobounds, timebounds = sightsurvey.getData(SIGHT_DATA, skip_rows=0)

    print 'data: ', data[:5]
    print 'geobounds: ', geobounds
    print 'timebounds: ', timebounds
```

CHAPTER TWELVE

GEOSETUP - PATHFINDER

The following are a series of functions defined in the Geosetup package module pathfinder that can be used for importing NOAA's pathfinder dataset.

Import:

```
import numpy as np
from netCDF4 import Dataset
import sys, os
import math
```

Covert coordinates to index position:

```
    def coord2idx(coord, deg_range, cell_size, limit_keyword):
        '''Round decimal degrees or minutes to a defined limit'''

        # check that coordinate is valid
        if (coord < -deg_range) or (coord > deg_range):
            print '\nBathymetric grid coordinate range incorrect. Exiting.\n'
            sys.exit()

        # Calculate index position from provided coordinate
        idx = (coord/cell_size)+(deg_range/cell_size/2)

        # Round index to integer value up or down depending on boundary
        if limit_keyword == 'min':
            idx = int(math.floor(idx))
        elif limit_keyword == 'max':
            idx = int(math.ceil(idx))
        else :
            print '\nCoordinate keyword invalid. Exiting.\n'
            sys.exit()

        # Calculate decimal coordinate at rounded index position
new_coord = (idx - (deg_range/cell_size/2))*cell_size

return new_coord, idx
```

Print summary output:

```
def summary(file_path):
    ''' Extract gebco bathymetry data summary.'''

    # Get Lat/Lon from first data file in list
    dataset = Dataset(file_path,'r')

    cols, rows = dataset.variables["dimension"]
    grid_w_deg, grid_h_deg = dataset.variables["spacing"]
    min_lon, max_lon = dataset.variables["x_range"]
    min_lat, max_lat = dataset.variables["y_range"]
    min_z, max_z = dataset.variables["z_range"]
    z = dataset.variables["z"][:5]
```

```
    dataset.close()

    print '\nGebco Bathymetric Data Summary:'
    print '------------------------------------'
    print 'cols: %i rows: %i' % (cols, rows)
    print 'grid width: %06.5f grid height: %06.5f' % (grid_w_deg, grid_h_deg)
    print 'min_lon: %5.1f max_lon: %5.1f' % (min_lon, max_lon)
    print 'min_lat: %5.1f max_lat: %5.1f' % (min_lat, max_lat)
    print 'min_z: %i max_z: %i' % (min_z, max_z)
    print 'First five z: ', z
```

getGebcoData:

```
def getGebcoData(file_path,min_lon,max_lon,min_lat,max_lat):
    '''Extract gebco bathymetric data from geographic bounds

       cell_size: decimal degree x & y dimension of grid cells
       The depth data is a 1-D array. Given its size, it is faster
       to use fancy indexing to extract the geographical subsection.
    '''
    dataset = Dataset(file_path,'r')
    cell_size = dataset.variables["spacing"][0]
    cols, rows = dataset.variables["dimension"]

    # Set lon (column) index and lat (row) index, retrieve adjusted coords
    min_lon, min_lon_idx = coord2idx(min_lon, 360, cell_size, 'min')
    max_lon, max_lon_idx = coord2idx(max_lon, 360, cell_size, 'max')
    min_lat, min_lat_idx = coord2idx(min_lat, 180, cell_size, 'min')
    max_lat, max_lat_idx = coord2idx(max_lat, 180, cell_size, 'max')

    # TODO check if incorrect to offset by one
    lon_range = (max_lon_idx - min_lon_idx) + 1
    lat_range = (max_lat_idx - min_lat_idx) + 1
    data_range = lon_range * lat_range

    zi = 0
    # Create zero array with the appropriate length for the data subset
    z = np.zeros(data_range)
    # Process number of rows for which data is being extracted
    for i in range((max_lat_idx - min_lat_idx)):
        # Pull row, then desired elements of that row into buffer
        tmp = (dataset.variables["z"][(i*cols):((i*cols)+cols)])[min_lon_idx:max_lon_idx]
        # Add each item in buffer sequentially to data array
        for j in tmp:
            z[zi] = j
            # Keep a count of what index position the next data point goes to
            zi += 1

    dataset.close()

    # Create latitude and longitude arrays
    lon_const = 360./cell_size/2
    lat_const = 180./cell_size/2
    lons = (np.asarray(range(lon_range)) + min_lon_idx - lon_const)*cell_size
    lats = (np.asarray(range(lat_range)) + min_lat_idx - lat_const)*cell_size
    lons, lats = np.meshgrid(lons,lats)
    lons = np.ravel(lons)
    lats = np.ravel(lats)

    # TODO remove
    print 'len_lons: ', lons.shape
    print 'len_lats: ', lats.shape
    print 'len_z:    ', z.shape

    return lons, lats, z
```

Main Program:

```
if __name__ == '__main__':

    # commandline usage
    if len(sys.argv) < 2:
        print >>sys.stderr,'Usage:',sys.argv[0],'<data directory>\n'
        sys.exit(1)

    data_dir = sys.argv[1]
    data_file = 'gridone.nc'

    file_path = os.path.join(data_dir,data_file)

    summary(file_path)

    # getGebcoData(file_path,min_lon,max_lon,min_lat,max_lat):
    lons, lats, z = getGebcoData(file_path,-20,20,-20,20)

    print 'lons: ', lons[:]
    print 'lats: ', lats[:]
    print 'z: ', z[:]
```

CHAPTER

THIRTEEN

GEOSETUP - CORTAD

As described by Trond Kristiansen on his blog [7]. The following are methods developed by him to import data from NOAA's CoRTAD data series.

Imports:

```
import os, sys, datetime, string
import numpy as np
from netCDF4 import Dataset
import numpy.ma as ma
import matplotlib.pyplot as plt
from pylab import *
import mpl_util
```

Get CoRTAD Time:

```
def getCORTADtime():

    #base="http://data.nodc.noaa.gov/thredds/dodsC/cortad/Version4/"
    base='/media/data/storage02/asf-fellowship/data/CoRTAD/version4/'
    file1="cortadv4_row00_col05.nc"
    filename1=base+file1
    cdf1=Dataset(filename1)

    print "Time: Extrating timedata from openDAP: %s"%(filename1)

    time=np.squeeze(cdf1.variables["time"][:])
    cdf1.close()
    return time
```

Open CoRTAD Files:

```
def openCoRTAD():
    # TODO Generalize to accept dates and geobounds, data dir
    """ Info on the different tiles used to identoify a region is found here:
    http://www.nodc.noaa.gov/SatelliteData/Cortad/TileMap.jpg"""
    #base='/media/data/storage02/asf-fellowship/data/CoRTAD/version4/'
    base= '/home/ryan/code/python/projects/asf/geosetup/data/cortad/'
#    base="http://data.nodc.noaa.gov/thredds/dodsC/cortad/Version4/"

    start_row = 00
    start_col = 05
    end_row = 01

    file1="cortadv4_row00_col05.nc"
    file2="cortadv4_row00_col06.nc"
    file3="cortadv4_row00_col07.nc"
    file4="cortadv4_row00_col08.nc"
    file5="cortadv4_row00_col09.nc"
    file6="cortadv4_row01_col05.nc"
    file7="cortadv4_row01_col06.nc"
    file8="cortadv4_row01_col07.nc"
```

```
        file9="cortadv4_row01_col08.nc"
        file10="cortadv4_row01_col09.nc"

        filename1=base+file1
        filename2=base+file2
        filename3=base+file3
        filename4=base+file4
        filename5=base+file5
        filename6=base+file6
        filename7=base+file7
        filename8=base+file8
        filename9=base+file9
        filename10=base+file10

        cdf1=Dataset(filename1)
        cdf2=Dataset(filename2)
        cdf3=Dataset(filename3)
        cdf4=Dataset(filename4)
        cdf5=Dataset(filename5)
        cdf6=Dataset(filename6)
        cdf7=Dataset(filename7)
        cdf8=Dataset(filename8)
        cdf9=Dataset(filename9)
        cdf10=Dataset(filename10)

        return cdf1,cdf2,cdf3,cdf4,cdf5,cdf6,cdf7,cdf8,cdf9,cdf10

    def extractCoRTADLongLat():
        """Routine that extracts the longitude and latitudes for the
        combination of tiles. This is only necessary to do once so it is separated
        from the extraction of SST."""
    #     cdf1,cdf2,cdf3,cdf4,cdf5,cdf6=openCoRTAD()
        cdf1,cdf2,cdf3,cdf4,cdf5,cdf6,cdf7,cdf8,cdf9,cdf10=openCoRTAD()

        longitude1=np.squeeze(cdf1.variables["lon"][:])
        latitude1=np.squeeze(cdf1.variables["lat"][:])

        longitude2=np.squeeze(cdf2.variables["lon"][:])
        latitude2=np.squeeze(cdf2.variables["lat"][:])

        longitude3=np.squeeze(cdf3.variables["lon"][:])
        latitude3=np.squeeze(cdf3.variables["lat"][:])

        longitude4=np.squeeze(cdf4.variables["lon"][:])
        latitude4=np.squeeze(cdf4.variables["lat"][:])

        longitude5=np.squeeze(cdf5.variables["lon"][:])
        latitude5=np.squeeze(cdf5.variables["lat"][:])

        longitude6=np.squeeze(cdf6.variables["lon"][:])
        latitude6=np.squeeze(cdf6.variables["lat"][:])

        longitude7=np.squeeze(cdf7.variables["lon"][:])
        latitude7=np.squeeze(cdf7.variables["lat"][:])

        longitude8=np.squeeze(cdf8.variables["lon"][:])
        latitude8=np.squeeze(cdf8.variables["lat"][:])

        longitude9=np.squeeze(cdf9.variables["lon"][:])
        latitude9=np.squeeze(cdf9.variables["lat"][:])

        longitude10=np.squeeze(cdf9.variables["lon"][:])
        latitude10=np.squeeze(cdf9.variables["lat"][:])

        cdf1.close();cdf2.close();cdf3.close();cdf4.close();cdf5.close();cdf6.close();cdf7.close();cdf8.clos

        longitude=concatenate((longitude1,longitude2,longitude3,longitude4,longitude5)) # 1-D array, axis ir
        latitude=concatenate((latitude6,latitude1)) # 1-D array, axis irrelevant

        """ We have to flip this array so that we have increasing latitude
         values required by np.interp function. This means we also have to
         flip the input SST array"""
```

```
#    latitude=np.flipud(latitude) # flipping doesn not appear to be necessary
    lons,lats=np.meshgrid(longitude,latitude)

    print "Extracted longitude-latitude for CoRTAD region"
    print "Long min: %s Long max: %s"%(longitude.min(),longitude.max())
    print "Lat min: %s Lat max: %s"%(latitude.min(),latitude.max())
    print "------------------------------\n"
    return lons, lats, longitude, latitude
```

Extract CoRTAD SST:

```
def extractCORTADSST(timestamp_start, timestamp_end, masked=True):
    """Routine that extracts the SST values for
    the specific tiles and time-period (t)"""
    cdf1,cdf2,cdf3,cdf4,cdf5,cdf6,cdf7,cdf8,cdf9,cdf10=openCoRTAD()

    # calculate days from ref date to first and last sighting
    ref_date=datetime.datetime(1980,12,31,12,0,0)
    days = 60.*60.*24. # sec*min*hr
    t1 = int(round((timestamp_start - ref_date).total_seconds()/days))
    t2 = int(round((timestamp_end - ref_date).total_seconds()/days))

    cortad_time=np.squeeze(cdf1.variables["time"][:])

    # use binary search to find index of nearest time value to data times
    idx1 = (np.abs(cortad_time-t1)).argmin()
    idx2 = (np.abs(cortad_time-t2)).argmin()

    # TODO make sure data is being averaged correctly
    filledSST1=np.average((cdf1.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST2=np.average((cdf2.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST3=np.average((cdf3.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST4=np.average((cdf4.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST5=np.average((cdf5.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST6=np.average((cdf6.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST7=np.average((cdf7.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST8=np.average((cdf8.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST9=np.average((cdf9.variables["FilledSST"][idx1:idx2,:,:]), axis=0)
    filledSST10=np.average((cdf10.variables["FilledSST"][idx1:idx2,:,:]), axis=0)

    offset=cdf1.variables["FilledSST"].__getattribute__('add_offset')
    cdf1.close();cdf2.close();cdf3.close();cdf4.close();cdf5.close();cdf6.close();cdf7.close();cdf8.close

    filledMaskedSST1=filledSST1 - offset
    filledMaskedSST2=filledSST2 - offset
    filledMaskedSST3=filledSST3 - offset
    filledMaskedSST4=filledSST4 - offset
    filledMaskedSST5=filledSST5 - offset
    filledMaskedSST6=filledSST6 - offset
    filledMaskedSST7=filledSST7 - offset
    filledMaskedSST8=filledSST8 - offset
    filledMaskedSST9=filledSST9 - offset
    filledMaskedSST10=filledSST10 - offset

#    filledMaskedSST1=filledSST3*0
#    filledMaskedSST2=filledSST3*0
#    filledMaskedSST3=filledSST3*0
#    filledMaskedSST4=filledSST3*0
#    filledMaskedSST5=filledSST3*0
#    filledMaskedSST6=filledSST3*0

    """Now we have all the data in 4 different arrays that we need to concentate.
    First we add the horisontal tiles, and finally we stack the two horisontal ones on top
    of each other."""
    filledMaskedSST_lower=concatenate((filledMaskedSST1,filledMaskedSST2,filledMaskedSST3,filledMaskedSS

    filledMaskedSST_upper=concatenate((filledMaskedSST6,filledMaskedSST7,filledMaskedSST8,filledMaskedSS

    filledMaskedSST_all=concatenate((filledMaskedSST_upper,filledMaskedSST_lower),axis=0)

    """Flip the SST array to be consistent with order of latitude array"""
 #   filledMaskedSST_all=np.flipud(filledMaskedSST_all) # flipping doesn not appear to be necessary
```

```

    """ Scale and offset is autmoatically detected and edited by netcdf, but
    we need to mask the values that are not filled."""
    filledMaskedSST_final=ma.masked_less(filledMaskedSST_all,-2.)

    print "Min and max of SST: %s - %s"%(filledMaskedSST_final.min(),filledMaskedSST_final.max())
    print "------------------------------\n"

    return filledMaskedSST_final
```

Main function:

```
if __name__ == "__main__":

    main()
```

CHAPTER

FOURTEEN

GEOSETUP - GLOBCOLOUR

The following are methods from the `globcolour` module in the `Geosetup` package for importing cholorphyll a data from the GlobColour data series produced by ESA.

14.1 Importing Mapped L3m Data

Extract GlobColour chlorophyll data from netCDFs

```
import sys, os
from netCDF4 import Dataset
import numpy as np
import numpy.ma as ma
import isingrid as grid
import re
import csv
import datetime
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
```

Subset geospatials data (remove?):

```
def subset_geodata(max_lon, min_lon, max_lat, min_lat,lons, lats, values):
    '''subset_geodata  returns a subset of lat/lon/value data from defined
       defined bounds in decimal degrees'''
    subset_lons = lons[(lons<max_lon)&(lons>min_lon)&(lats<max_lat)&(lats>min_lat)]
    subset_lats = lats[(lons<max_lon)&(lons>min_lon)&(lats<max_lat)&(lats>min_lat)]
    subset_vals = values[(lons<max_lon)&(lons>min_lon)&(lats<max_lat)&(lats>min_lat)]
    return subset_lons, subset_lats, subset_vals
```

Find nearest value in array, and return the index position of it:

```
def find_nearest(array,value):
    '''
    Returns index position of element nearest to given value
    '''
    idx = (np.abs(array-value)).argmin()
    #return array[idx] # return value
    return idx # return index
```

Get Mapped GlobColour Data:

```
def getMappedGlob(data_dir, min_lon, max_lon, min_lat, max_lat, data_time_start, data_time_end):
    '''
    Extracts subset of globcolour data based on lat/lon bounds and dates in the
    iso format '2012-01-31'
    '''

```

```
    #TODO figure out if this is right
    file_list = np.asarray(os.listdir(data_dir))
    file_dates = np.asarray([datetime.datetime.strptime(re.split('[._]', filename)[1], '%Y%m%d')
                             for filename in file_list])
    data_files = file_list[(file_dates >= data_time_start) & (file_dates <= data_time_end)]

    # Get Lat/Lon from first data file in list
    dataset = Dataset(os.path.join(data_dir,file_list[0]),'r')
    lons = dataset.variables["lon"][:]
    lats = dataset.variables["lat"][:]
    vals_sum = np.zeros_like(dataset.variables["CHL1_mean"][:][:])
    mask_sum = np.empty(np.shape(vals_sum),dtype=bool)
    dataset.close()

    # Create cumulative mask before averaging data
    for data_file in data_files:
        current_file = os.path.join(data_dir,data_file)
        dataset = Dataset(current_file,'r') # by default numpy masked array
        mask = (dataset.variables["CHL1_mean"][:][:]).mask
        mask_sum =  mask + mask_sum

    vals_sum.mask = mask_sum

    # Get average of chlorophyl values from date range
    file_count = 0
    for data_file in data_files:
        current_file = os.path.join(data_dir,data_file)
        dataset = Dataset(current_file,'r') # by default numpy masked array
        vals = dataset.variables["CHL1_mean"][:][:]
        vals.mask = mask_sum
        vals_sum += vals
        #TODO remove
        #printstuff(vals,vals_sum,file_count)
        file_count += 1
        dataset.close()

    vals_mean = vals/file_count #TODO simple average

    # Mesh Lat/Lon the unravel to return lists
    #TODO make mesh optional (i.e. return grid or lists)
    lons_mesh, lats_mesh = np.meshgrid(lons,lats)
    lons = np.ravel(lons_mesh)
    lats = np.ravel(lats_mesh)

    # Print Globcolour Information
    print '\nChl-a Information'
    print '---------------------------------------------'
    print 'Start Date: ', data_time_start
    print 'End Date: ', data_time_end
    print 'Max Chl-a: ', np.amax(vals_mean)
    print 'Min Chl-a: ', np.amin(vals_mean)

    return lons, lats, vals_mean
```

Main Program:

```
if __name__ == '__main__':

    # commandline usage #
    if len(sys.argv) < 2:
        print >>sys.stderr,'Usage:',sys.argv[0],'<data directory>\n'
        sys.exit(1)

    data_dir = sys.argv[1]

    lons, lats, vals_mean = getMappedGlobcolour(data_dir,0.,50.,30.,60.,'2007-08-01','2007-08-30')
    print lons
```

14.2 Importing ISIN grid L3b Data

The following methods can be used to convert data in the ISIN grid format into latitude and longtitude data, which is a Python translation of the R code that was posted on the following blog page:

http://menugget.blogspot.com/2012/04/working-with-globcolour-data.html

Import:

```python
import numpy as np

'''
from: http://menugget.blogspot.no/2012/04/working-with-globcolour-data.html#more

This function is used converts ISIN grid information used by Globcolour to latitude
and longitude for a pefect sphere, as well as to construct associated polygons for use in mapping.

The raw Globcolour .nc files come with column and row pointers
as to the the grid's location. For 4.63 km resolution data, this
translates to 4320 latitudinal rows with varying number of
associated longitudinal columns depending on the latitudinal
circumference.

Input must be either a vector of grid numbers ["grd"] or a dataframe
with column and row identifiers ["coord", e.g. columns in coord$col and
rows in coord$row]

When the argument "polygon=FALSE" (Default), the function will output
a dataframe object containing grid information (gridnumber["output$grd"],
column["output$col"], row["output$row"], longitude["output$lon"], and
latitude[output$lat])

If the argument "polygon=TRUE", then the putput will be a list with
polygon shapes in a dataframe(longitudinal coordinates of corners
["[[i]]$x"], latitudinal coordinates of corners ["[[i]]$y"])
'''
```

Convert ISIN grid to Lat Lon Coordinates:

```python
def isin_convert(grid = None, coord = None, polygons = False):

    earth_radius = 6378.137 # TODO this was different in globcolour metadata
    Nlat = 4320 # Number of latitudinal bands, globcolour default 4km resolution
    pi = np.pi
    circum = 2*pi*earth_radius # circumference of Earth at equator

    lat_rows = np.arange(1,Nlat+1,1) # nparray of sequence 1 to Nlat

    # circumfrance at equator / lat bin width:
    # delta-radius, varies by how many parallels there are (Nlat)
    dr = (pi*earth_radius)/Nlat
    # angle increment between each parrallel:
    # delta-phi, from equator to pole
    dphi_lat = pi/Nlat

    phi = -(pi/2)+(lat_rows*dphi_lat)-(dphi_lat/2) # nparray of angles
    # calculate latitudal circumference of parallels:
    p = circum*np.cos(phi) # np array of circ in __ (2*Pi*r)*cos(phi)
    # number of lon rows, dimension = lat dimension at equ.
    Nlon = np.rint(p/dr) #round to nearest whole integer
    dlon = p/Nlon # circumfrance / number of meridians
    dphi_lon = (2*pi)/Nlon
    Ntot = sum(Nlon)
    lat = phi*(180/pi) # nparray of angles to radians

    if (grid is not None):
        # calculate coordinates
```

```
        cum_Nlon = np.cumsum(Nlon)
        # TODO the following doesn't work for array [1,2,3,4], matters?
        grid_rows = np.asarray([np.amax(np.where(cum_Nlon < point)) for point in grid])
        grid_cols = grid - cum_Nlon[grid_rows]
        # calculate longitude and latitude
        grid_lats = lat[grid_rows]
        Nlon_rows = Nlon[grid_rows]
        grid_lons = (360*(grid_cols-0.5)/Nlon_rows)-180

    if (coord is not None):
        #calculate coordinates
        grid_rows = coord[:][0]
        grid_cols = coord[:][1]
        print grid_rows
        print grid_cols
        # calcuate longitude and latitude
        grid_lats = lat[grid_rows]
        Nlon_rows = Nlon[grid_rows]
        grid_lons = (360*(grid_cols-0.5)/Nlon_rows)-180
        # calculate grid
        cum_Nlon = np.cumsum(Nlon)
        grid = cum_Nlon[grid_rows] + grid_cols # TODO check what this does
        cum_Nlon = np.cumsum(Nlon)
        return  grid_lats, grid_lons
    if (polygons is not None):
        Nlon_rows = Nlon[grid_rows]
        grid_width = 360/Nlon_rows
        grid_height = 180/Nlat
        # create list 1 to (number of elements in grid)
        polys = range(0,len(grid))
        xs = np.hstack([grid_lons[polys]-grid_width/2,
                        grid_lons[polys]-grid_width/2,
                        grid_lons[polys]+grid_width/2,
                        grid_lons[polys]+grid_width/2])

        ys = np.hstack([grid_lats[polys]-grid_height/2,
                        grid_lats[polys]+grid_height/2,
                        grid_lats[polys]+grid_height/2,
                        grid_lats[polys]-grid_height/2])
#TODO check that what's retu
#               return xs,ys #np.vstack([xs,ys])

    if (polygons is None):
        array = np.vstack([grid,grid_cols,grid_rows,grid_lons,grid_lats])
```

Main function:

```
if __name__ == "__main__":
    grid = np.array([20,35,60,900])
    print isin_convert(grid)
```

CHAPTER FIFTEEN

GEOSETUP - GEBCO

The methods in the `gebco` module of the `Geosetup` package allow for importing of the GEBCO Bathymetric data provided by the British Oceanographic Data Centre.

Imports:

```
import numpy as np
from netCDF4 import Dataset
import sys, os
import math
```

Convert coordinates to index positions:

```
def coord2idx(coord, deg_range, cell_size, limit_keyword):
    '''Round decimal degrees or minutes to a defined limit'''

    # check that coordinate is valid
    if (coord < -deg_range) or (coord > deg_range):
        print '\nBathymetric grid coordinate range incorrect. Exiting.\n'
        sys.exit()

    # Calculate index position from provided coordinate
    idx = (coord/cell_size)+(deg_range/cell_size/2)

    # Round index to integer value up or down depending on boundary
    if limit_keyword == 'min':
        idx = int(math.floor(idx))
    elif limit_keyword == 'max':
        idx = int(math.ceil(idx))
    else :
        print '\nCoordinate keyword invalid. Exiting.\n'
        sys.exit()

    # Calculate decimal coordinate at rounded index position
    new_coord = (idx - (deg_range/cell_size/2))*cell_size

    return new_coord, idx
```

Print summary info:

```
def summary(file_path):
    ''' Extract gebco bathymetry data summary.'''

    # Get Lat/Lon from first data file in list
    dataset = Dataset(file_path,'r')

    cols, rows = dataset.variables["dimension"]
    grid_w_deg, grid_h_deg = dataset.variables["spacing"]
    min_lon, max_lon = dataset.variables["x_range"]
    min_lat, max_lat = dataset.variables["y_range"]
    min_z, max_z = dataset.variables["z_range"]
    z = dataset.variables["z"][:5]
```

```
    dataset.close()

    print '\nGEBCO Bathymetric Data Summary:'
    print '------------------------------------'
    print 'cols: %i rows: %i' % (cols, rows)
    print 'grid width: %06.5f grid height: %06.5f' % (grid_w_deg, grid_h_deg)
    print 'min_lon: %5.1f max_lon: %5.1f' % (min_lon, max_lon)
    print 'min_lat: %5.1f max_lat: %5.1f' % (min_lat, max_lat)
    print 'min_z: %i max_z: %i' % (min_z, max_z)
    print 'First five z: ', z
```

Get GEBCO data:

```
def getGEBCOData(file_path,min_lon,max_lon,min_lat,max_lat):
    '''Extract gebco bathymetric data from geographic bounds

       cell_size: decimal degree x & y dimension of grid cells
       The depth data is a 1-D array. Given its size, it is faster
       to use fancy indexing to extract the geographical subsection.
    '''
    dataset = Dataset(file_path,'r')
    cell_size = dataset.variables["spacing"][0]
    cols, rows = dataset.variables["dimension"]

    # Set lon (column) index and lat (row) index, retrieve adjusted coords
    min_lon, min_lon_idx = coord2idx(min_lon, 360, cell_size, 'min')
    max_lon, max_lon_idx = coord2idx(max_lon, 360, cell_size, 'max')
    min_lat, min_lat_idx = coord2idx(min_lat, 180, cell_size, 'min')
    max_lat, max_lat_idx = coord2idx(max_lat, 180, cell_size, 'max')

    # TODO remove
    print 'min_lon_idx', min_lon_idx, 'min_lon', min_lon
    print 'max_lon_idx', max_lon_idx, 'max_lon', max_lon
    print 'min_lat_idx', min_lat_idx, 'min_lat', min_lat
    print 'max_lat_idx', max_lat_idx, 'max_lat', max_lat

    # TODO check if incorrect to offset by one
    lon_range = (max_lon_idx - min_lon_idx) + 1
    lat_range = (max_lat_idx - min_lat_idx) + 1
    data_range = lon_range * lat_range

    zi = 0
    # Create zero array with the appropriate length for the data subset
    z = np.zeros(data_range)
    # Process number of rows for which data is being extracted
    for i in range((max_lat_idx - min_lat_idx)):
        # Pull row, then desired elements of that row into buffer
        tmp = (dataset.variables["z"][(i*cols):((i*cols)+cols)])[min_lon_idx:max_lon_idx]
        # Add each item in buffer sequentially to data array
        for j in tmp:
            z[zi] = j
            # Keep a count of what index position the next data point goes to
            zi += 1

    dataset.close()

    # Create latitude and longitude arrays
    lon_const = 360./cell_size/2
    lat_const = 180./cell_size/2
    lons = (np.asarray(range(lon_range)) + min_lon_idx - lon_const)*cell_size
    lats = (np.asarray(range(lat_range)) + min_lat_idx - lat_const)*cell_size
    lons, lats = np.meshgrid(lons,lats)
    lons = np.ravel(lons)
    lats = np.ravel(lats)

    # TODO remove
    print 'len_lons: ', lons.shape
    print 'len_lats: ', lats.shape
    print 'len_z:    ', z.shape

    return lons, lats, z
```

Main Program:

```
if __name__ == '__main__':

    ####################
    # commandline usage #
    ####################
    if len(sys.argv) < 2:
        print >>sys.stderr,'Usage:',sys.argv[0],'<data directory>\n'
        sys.exit(1)
    data_dir = sys.argv[1]
    data_file = 'gridone.nc'

    file_path = os.path.join(data_dir,data_file)

    summary(file_path)

    # getGEBCOData(file_path,min_lon,max_lon,min_lat,max_lat):
    lons, lats, z = getGEBCOData(file_path,-20,20,-20,20)

    print 'lons: ', lons[:]
    print 'lats: ', lats[:]
    print 'z: ', z[:]
```

CHAPTER

SIXTEEN

GEOSETUP - MAIN PROGRAM

The following is the main program script for the `Geosetup` package that will import data from a comma separated text file and automatically produce sea surface temperature, chlorophyll a, and bathymetry datasets bound by the geographic bounds of the data provided.

Import:

```
#/usr/bin/env python

from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np
import sys, os, errno
import re
import math
import datetime
import numpy.lib.recfunctions
import pyproj
from StringIO import StringIO
import geosetup.mpl_util
#TODO make module naming consistent
from geosetup.globcolour import globcolour
from geosetup.pathfinder import pathfinder
from geosetup.gebco import gebco
from geosetup.interpolate import invdistgis, invdist
from geosetup.interpolate.datainterp import geointerp
from geosetup.writefile import data2raster
from geosetup.sightsurvey import sightsurvey

# prevent creation of .pyc files
sys.dont_write_bytecode = True
```

```
def plotSizedData(map_obj,lons,lats,values,symbol,min_size,max_size,lines=False):
    '''
    Plot data with varying sizes
    '''
    proj_x,proj_y = m(lons,lats)

    # Calculate marker sizes using y=mx+b
    # where y = marker size and x = data value
    slope = (max_size-min_size)/(max(values)-min(values))
    intercept = min_size-(slope*min(values))

    for x, y, val in zip(proj_x, proj_y, values):
        msize = (slope*val)+intercept
        map_obj.plot(x, y, symbol, markersize=msize)
```

```
def plotLines(map_obj,lons,lats,lw=1.0,color='k'):
    '''
    Draw a line between each set of coordinates
    '''
```

```
    for i in range(len(lons)-1):
        map_obj.drawgreatcircle(lons[i],lats[i],lons[i+1],lats[i+1],linewidth=1.0,color='k')
```

```
def centerMap(lons,lats,scale):
    '''
    Set range of map. Assumes -90 < Lat < 90 and -180 < Lon < 180, and
    latitude and logitude are in decimal degrees
    '''
    north_lat = max(lats)
    south_lat = min(lats)
    west_lon = max(lons)
    east_lon = min(lons)

    # find center of data
    # average between max and min longitude
    lon0 = ((west_lon-east_lon)/2.0)+east_lon

    # define ellipsoid object for distance measurements
    g = pyproj.Geod(ellps='WGS84') # Use WGS84 ellipsoid TODO make variable
    earth_radius = g.a # earth's radius in meters

    # Use pythagorean theorom to determine height of plot
    # divide b_dist by 2 to get width of triangle from center to edge of data area
    # inv returns [0]forward azimuth, [1]back azimuth, [2]distance between

    # a_dist = the height of the map (i.e. mapH)
    b_dist = g.inv(west_lon, north_lat, east_lon, north_lat)[2]/2
    c_dist = g.inv(west_lon, north_lat, lon0, south_lat)[2]

    mapH = pow(pow(c_dist,2)-pow(b_dist,2),1./2)
    lat0 = g.fwd(lon0,south_lat,0,mapH/2)[1]

    # distance between max E and W longitude at most southern latitude
    mapW = g.inv(west_lon, south_lat, east_lon, south_lat)[2]

    return lon0, lat0, mapW*scale, mapH*scale
```

```
def drawsst(ax, map_object, longSST, latSST, filledSST, myalpha):
    '''Draw Sea surface temperature

    Author - Trond Kristiansen'''

    # Input arrays have to be 2D
    print "Drawing SST: max %s and min %s"%(filledSST.min(), filledSST.max())
    x2, y2 = map_object(longSST,latSST)
    levels=np.arange(2,18,0.5)

    # TODO correct issue with colormap
    if myalpha > 0.99:
        CS2 = map_object.contourf(x2, y2, filledSST, levels,
                #cmap=mpl_util.LevelColormap(levels,cmap=cm.RdYlBu_r),
                cmap = plt.cm.jet,
                extend = 'upper', alpha=myalpha)
    else:
        CS2 = map_object.contourf(x2, y2, filledSST, levels,
                #cmap=mpl_util.LevelColormap(levels,cmap=cm.RdYlBu_r),
                cmap = plt.cm.jet,
                extend = 'upper', alpha=myalpha)

        # TODO correct or remove
        #CS2 = map_object.contourf(x2,y2,filledSST,levels,
        #                cmap=mpl_util.LevelColormap(levels,cmap=cm.Greys),
        #                extend='upper',alpha=myalpha)
```

```
def find_nearest(array, value):
    '''
    TODO
    '''
    idx = (np.abs(array - value)).argmin()
    #return array[idx] # return value
    return idx # return index
```

```python
def filter2bool(regexp, array):
    '''
    Create array of positive boolean where elements match regex
    '''
    return np.array([bool(re.search(regexp, element)) for element in array])

if __name__ == '__main__':

    ####################
    # Commandline Usage #
    ####################

    if len(sys.argv) < 3:
        print >>sys.stderr,'\nUsage:',sys.argv[0],'<datafile> <#rows to skip>\n'
        sys.exit(1)

    ############################
    # Configuration parameters #
    ############################

    PROJ_DIR = '/home/ryan/Desktop/asf-fellowship/code/geosetup/'
    SST_DIR = 'data/pathfinder/'
    CHL_DIR = 'data/globcolour/'
    BTM_DIR = 'data/gebco/gridone.nc'
    SIGHT_DATA = sys.argv[1] # 'data/survey/na07.tab'
    OUT_DIR = 'output/'
    #TODO incorporate regrid size
    GRD_SIZE = 50 #km or deg?

    # Test that output directory exists, if not create it
    try:
        os.makedirs(OUT_DIR)
    except OSError, e:
        if e.errno != errno.EEXIST:
            raise
```

```python
    #########################
    # Process Sighting Data #
    #########################

    # Get sight surveying data and geographical and time bounds for data
    data = sightsurvey.getData(SIGHT_DATA, skip_rows=0)

    # Get date information for subsampling environment data
    datetimes = list()
    for data_date,data_time in zip(data['dates'],data['times']):
        data_datetime = data_date+data_time
        datetimes.append(datetime.datetime.strptime(data_datetime,'%y%m%d%H%M%S'))

    LON_START = min(data['lon'])
    LON_END = max(data['lon'])
    LAT_START = min(data['lat'])
    LAT_END = max(data['lat'])
    TIME_START = min(datetimes)
    TIME_END = max(datetimes)

    # Print sighting data informtion
    print 'Sighting Period: ', TIME_START, TIME_END, TIME_END - TIME_START
```

```python
    ###############
    # Create Grid #
    ###############

    grid_lat_start =  math.floor(LAT_START)
    grid_lon_start =  math.floor(LON_START)
    grid_lat_end =  math.ceil(LAT_END)
    grid_lon_end =  math.ceil(LON_END)
    grid_lats = np.linspace(GRD_SIZE, 180, 180/GRD_SIZE)
    grid_lons = np.linspace(GRD_SIZE, 180, 180/GRD_SIZE)
```

```
        #######################################
        # Calculate Sightings per unit effort #
        #######################################
        # 'BM' Blue Whale (Balaenoptera musculus)
        # 'BP' Fin Whale (Balaenoptera physalus)
        # 'BB' Sei Whale (Balaenoptera borealis)
        # 'BA' Minke whales (Balaenoptera acutorostrata)
        # 'MN' Humpback Whale (Megaptera novaeangliae)

        # Create array of indexes for minke whales
        # TODO verify the effort calc is correct
        minke_idx = np.where(filter2bool('BA',data['species'])==True)[0]
        minke_effort = data['effort_nmil'][minke_idx]
        minke_lat = data['lat'][minke_idx]
        minke_lon = data['lon'][minke_idx]

        # Create Effort Gtiff
        effortGeopoint = data2raster.GeoPoint(minke_lon, minke_lat, minke_effort)
        effortGeopoint.create_raster(filename = OUT_DIR + "effort.tiff", output_format="GTiff")

        # Interpolate / Plot Minke effort
        minke_x, minke_y = effortGeopoint.transform_point()
        ZI = invdist.invDist(minke_lat,minke_lon, minke_effort)

        XI, YI = np.meshgrid(minke_x, minke_y)
        n = plt.normalize(0.0, 1000.0)
        plt.subplot(1, 1, 1)
        plt.pcolor(XI, YI, ZI)
        #plt.scatter(xv, yv, 100, values)
        plt.colorbar()
        plt.show()
```

```
        ######################
        # Process CorTAD Data # TODO revue / remove
        ######################

    #     # Get cortad SST within date period
    #     # TODO modify method to subset lat/lon
    #     filledSST = cortad.extractCORTADSST("North Sea", TIME_START, TIME_END)
    #     lonSST2D, latSST2D, sst_lon, sst_lat = cortad.extractCoRTADLongLat()
    #     sst_lon, sst_lat = np.meshgrid(sst_lon,sst_lat)
    #     sst_lon = np.ravel(sst_lon)
    #     sst_lat = np.ravel(sst_lat)
    #     filledSST_flat = np.ravel(filledSST)

    #     # Create SST Gtiff
    #     sstGeopoint = data2raster.GeoPoint(sst_lon, sst_lat, filledSST_flat)
    #     sstGeopoint.create_raster(filename = OUT_DIR + "sst.tiff", output_format="GTiff")
```

```
        ###############################
        # Process Pathfinder SST Data #
        ###############################

        # Extract Chl-a datai
        # getnetcdfdata(data_dir, nc_var_name, min_lon, max_lon, min_lat, max_lat, data_time_start, data
        sst_lons, sst_lats, sst_vals = pathfinder.getnetcdfdata(PROJ_DIR + SST_DIR,
                                                                 'sea_surface_temperature',
                                                                  LON_START, LON_END,
                                                                  LAT_START, LAT_END,
                                                                  TIME_START, TIME_END)
        # Create SST Gtiff
        sstGeopoint = data2raster.GeoPoint(sst_lons, sst_lats, sst_vals)
        sstGeopoint.create_raster(filename = OUT_DIR + "sst.tiff", output_format="GTiff",
                                  cell_width_meters = 5000, cell_height_meters = 5000)
```

```
        #################################
        # Process Globcolour Chl-a Data #
        #################################

        # Extract Chl-a data
        chla_lons, chla_lats, chla_vals = globcolour.getMappedGlob(PROJ_DIR+CHL_DIR,
```

```
                                                LON_START, LON_END,
                                                LAT_START, LAT_END,
                                                TIME_START, TIME_END)
chla_vals = np.ravel(chla_vals)

# Create Chla Gtiff
chlaGeopoint = data2raster.GeoPoint(chla_lons,chla_lats,chla_vals)
chlaGeopoint.create_raster(filename = OUT_DIR + "chla.tiff", output_format="GTiff",
                           cell_width_meters = 5000, cell_height_meters = 5000)
```

```
############################
# Process Bathymetric Data #
############################

# Extract bathymetric data
bathy_lons, bathy_lats, bathy_z = gebco.getGebcoData(BTM_DIR, LON_START, LON_END,
                                                     LAT_START, LAT_END)
# Create Bathy Gtiff
bathyGeopoint = data2raster.GeoPoint(bathy_lons,bathy_lats,bathy_z)
bathyGeopoint.create_raster(filename = OUT_DIR + "bathy.tiff",output_format="GTiff")
```

```
###############
# Create Plot #
###############

fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(111)

# Calculate map's center lat and lon from sampling data
lon0center,lat0center,mapWidth,mapHeight = centerMap(data['lon'],data['lat'],1.1)

# Lambert Conformal Projection Plot
# lat_1 is first standard parallel.
# lat_2 is second standard parallel (defaults to lat_1).
# lon_0,lat_0 is central point.
# TODO check that following isn't more accurate from pyProj
# rsphere=(6378137.00,6356752.3142) specifies WGS4 ellipsoid
# area_thresh=1000 means don't plot coastline features less
# than 1000 km^2 in area.
m = Basemap(width=mapWidth,height=mapHeight,
            rsphere=(6378137.00,6356752.3142),\
            resolution='l',area_thresh=1000.,projection='lcc',\
            lat_1=45.,lat_2=55.,\
            lat_0=lat0center,lon_0=lon0center)

# Plot STT
#drawsst(ax,m,lonSST2D,latSST2D,filledSST,0.25)

# Plot Chl-a
#TODO
#drawchla()

# Plot Depth
#TODO
#drawbottom()

# Draw parallels and meridians.
m.drawparallels(np.arange(-80.,81.,20.), labels=[1,0,0,0], fontsize=10)
m.drawmeridians(np.arange(-180.,181.,20.), labels=[0,0,0,1], fontsize=10)
m.drawmapboundary(fill_color='aqua')
m.drawcoastlines(linewidth=0.2)
m.fillcontinents(color='white', lake_color='aqua')

# Plot data points
plotSizedData(m,minke_lon,minke_lat,minke_effort,'ro',4,20)
plotLines(m,data['lon'],data['lat'],lw=1.0,color='k')
#x, y = m(data['lon'],data['lat'])
#m.scatter(x,y,2,marker='o',color='k')

# Print Plot Information
print '\nPlot Information'
```

```
        print '-------------------------------------------'
        print "Map lat/lon center: ",lat0center,lon0center
        print "Map height/width: ",mapWidth,mapHeight
        # TODO write metadata to image file
        # http://stackoverflow.com/questions/10532614/can-matplotlib-add-metadata-to-saved-figures
        #plt.title("Example")
        #plotfile='SST_northsea_'+str(currentDate)+'.png'
        #print "Saving map to file %s"%(plotfile)
        #plt.savefig(plotfile)

        plt.show()
```

CHAPTER

SEVENTEEN

APPENDIX: TEMPORAL COVERAGE OF OCEAN COLOR SATELLITE MISSIONS

17.1 Missions

- SeaWiFS
- MODIS
- MERIS
- Pathfinder (merged)
- CoRTAD (merged)
- GlobColour (merged)

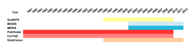


Figure 17.1: Temporal ranges of ocean color satellite missions

References

BIBLIOGRAPHY

[1] Ravish Bapna. *Participatory Geospatial Development Using Python*. Indian Institute of Technology, Kanpur, India, September 2012.

[2] Douglas Bates, John Chambers, Seth Falcon, Robert Gentleman, Kurt Hornik, Stefano Iacus, Ross Ihaka, Friedrich Leisch, Uwe Ligges, Thomas Lumley, Martin Maechler, Duncan Murdoch, Paul Murrel, Martyn Plummer, Brian Ripley, Deepayan Sarkar, Duncan Temple Lang, Luke Tierney, and Simon Urbanek. R - statistical programming language. 2013.

[3] Scott Chacon. *Pro Git*. Apress, Berkeley, California, 2009.

[4] Gerald I. Evenden. Cartographic projection procedures for the UNIX environment - a user's manual. Open-file Report, United States Department of the Interior Geological Survey, Woods Hole, MA, May 1990.

[5] William A. Huber. What's the best projection for rasterization of random latitude and longitude data in the northern atlantic? February 2013.

[6] Tim Kosse. Filezilla. 2013.

[7] Trond Kristiansen. Using python, openDAP, and matplotlib to plot CoRTAD SST on maps \textbar trond kristiansen. 2012.

[8] Jan Krymmel. UTM zones. 2007.

[9] Kurt Lambeck. *The earth's variable rotation: geophysical causes and consequences*. Cambridge University Press, Cambridge; New York, 1980.

[10] Mike Ruth. GeoTIFF FAQ version 2.4. 2011.

[11] Erik Westra. *Python Geospatial Development: Build a complete and sophisticated mapping application using Python tools for GIS development*. Packt Publishing, Birmingham-Mumbai, 1 edition, December 2010.

[12] Jeffery Whitaker. NetCDF4-Python - Python/Numpy interface to NetCDF. 2013.

[13] British Oceanographic Data Center. GEBCO gridded bathymetry data. 2013.

[14] Brockmann Consult. BEAM - earth observation toolbox and development platform. 2013.

[15] Google. Map types - google maps JavaScript API v3 — google developers. 2013.

[16] National Center for Atmospheric Research. NASA satellite product levels. 2013.

[17] Oak Ridge National Laboratory DAAC. ASCII grid format description. 2013.

[18] Unidata. NetCDF. 2013.

[19] GDAL. GDAL raster formats. 2013.

[20] HDF Group. HDF group products. 2013.

[21] MathWorks. Matlab. 2013.

[22] NOAA. NODC-UNC CoRTAD. 2013.

[23] NOAA. NODC pathfinder SST data. 2013.