OpenPolarServer An open source, field deployable, spatial data infrastructure.

Kyle W. Purdon, KU Geography
Trey Stafford, KU Geography
Sam Buchanan, KU Engineering
Haiji Wang, KU Engineering
John Paden, KU, CReSIS
Xingong Li, KU Geography

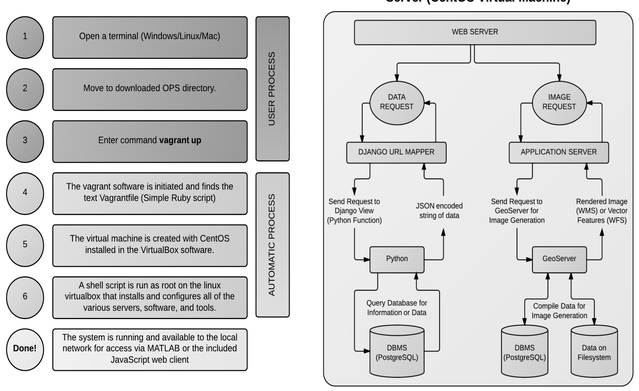
Coming Soon!
ops.cresis.ku.edu

What is the OpenPolarServer?

The OpenPolarServer (OPS) system is a packaged web server (Apache) hosting a database management system (PostgreSQL + PostGIS), map server (GeoServer), and a database-access API (Django) for **storing, retrieving, creating, and querying multi-layer polar remote sensing data**. The system is accessed via the MATLAB data picker or the JavaScript (GeoEXT + OpenLayers) web interface. The OPS is designed to be easily deployed in the field.

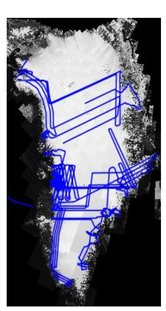
Deployable Web Server

A requirement of the OPS system is that it be **portable** for easy deployment in **field applications** where network access is limited or non-existent. To fulfill this requirement the system is packaged completely on a **CentOS (Linux) virtual machine** and deployed using a provisioning tool called Vagrant. To install the complete system a user must install Oracle VirtualBox and Vagrant, then download the OPS package. A simple command, [vagrant up] will automatically create and install the system, including all software and configuration. The system can be temporarily suspended and resumed using [vagrant suspend] and [vagrant resume] at any time. Vagrant also allows for a complete wipe of the system using [vagrant destroy]. The entire installation process currently takes around 15 minutes to complete. Once installed the user can interface with the system using MATLAB (installed separately) or the web interface. The vagrant process (left) and system structure (right) are diagramed below.



Map Server

A crucial piece of any spatial data infrastructure is a map server. The OPS system uses **GeoServer** in an **Apache Tomcat** container to serve WMS (Web Map Service) and WFS (Web Feature Service) data to the various clients. Typically GeoTIFF data (satellite **imagery**, **velocity** rasters, and **bedrock** grids) are served as WMS layers. Data collection tracks (**flightlines**) are served as either a **WMS** or a **WFS**. These services are queryable and therefore can produce dynamically generated maps including **real-time updated data** from the DBMS. The images below show a rendered JPEG (left) combining GeoTIFF imagery with line paths from the PostgreSQL database, and an OpenLayers application (right) showing information stored in the database for a selected line path with additional JavaScript tools.



greenland_natural_90m

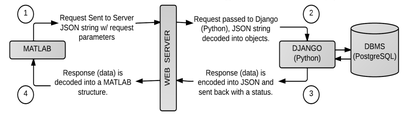
fid	RED_BAND	GREEN_BAND	BLUE_BAND
255.0	249.0	251.0	

arctic_rds_line_paths

fid	season_id	radar_id	segment_name	start_gps_time	stop_gps_time	location_id	season_name
arctic_rds_line_paths.13	1	1	20130415_01	1366023376.7803938	1366048373.3979449	1	2013_Greenland_P3

Database-Access API

Just as one must access the contents of a file to analyze the underlying values a DBMS must have an **API (Application Programming Interface)** to access and analyze its contents effectively. The OPS system uses the **Django ORM** (Object-relational mapper) to insert, retrieve, and query stored data. Django is built in **Python** and provides efficient and convenient shortcuts for interfacing with the database from various web clients. The database structure is defined in python using classes to map tables and class variables to define fields within the tables. The locations and radars classes for the CReSIS radar depth sounder are shown below (bottom). Django 'views' are defined as functions in python. These views are mapped to URL's which can be accessed from MATLAB or the web. A typical request cycle from MATLAB is outlined in the diagram below (top).




```
class locations(models.Model):
    location_id = models.AutoField(primary_key=True)
    location_name = models.CharField(max_length=20)

    def __unicode__(self):
        return "%s"%(self.location_name)
```

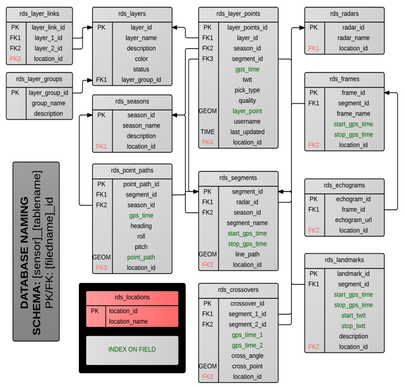
```
class radars(models.Model):
    radar_id = models.AutoField(primary_key=True)
    radar_name = models.CharField(max_length=20)
    location = models.ForeignKey('locations')

    def __unicode__(self):
        return "%s"%(self.radar_name)
```

PostGIS (Geospatial DBMS)

PostgreSQL alone cannot handle geospatial data in a **GIS friendly** format. However with the PostGIS **spatial database extender** geographic objects and methods are added to the functionality of the database. Objects like lines, points, and polygons can be stored in **geometry types** that include XY (2D), XYZ (3D), and XYZM (4D) objects. Having objects with understood spatial coordinates also allows for a suite of GIS functionality, which is included with PostGIS. With this extension queries can now ask questions like "what is the difference in values at the crossing of these two lines" (**crossover analysis**) and even "give me all of the available data inside of this boundary" (**geographic search**). PostGIS includes both geographic and projected coordinate system information using EPSG codes. In the OPS data is stored in EPSG 4326 (WGS 1984), then transformed into EPSG 3013/3413 (Polar Stereographic South/North). PostGIS includes functions for forward and backward projection.

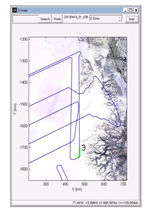
Database Management System

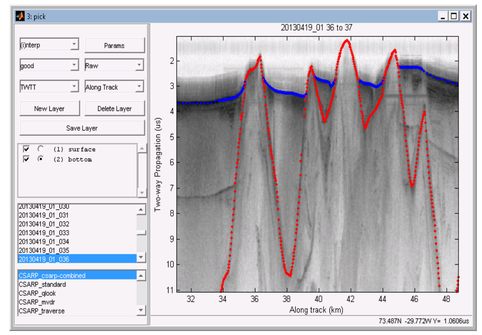
Previous to the OPS system most datasets were stored on the file system in various container formats such as HDF, NetCDF, or MAT. The limit of these file formats is that to make any inferences about the information they contain you must first load them, and then run some process to understand them. A database management system (DBMS) allows any structured dataset to be stored in parts (tables) and related via keys. Using the **structured query language** (SQL) this data can be instantly queried, making simple analysis effortless. The complication of using a DBMS is the initial set-up time and relational design. The **ER (Entity Relationship) diagram** for CReSIS radar data is shown below. This diagram outlines where data is stored in the database and how it is related to other data via keys. The database stores paths (latitude, longitude, time), layers (surface, bottom, internal), and layer data as well as reference information and data used for error analysis.



MATLAB Data Picker

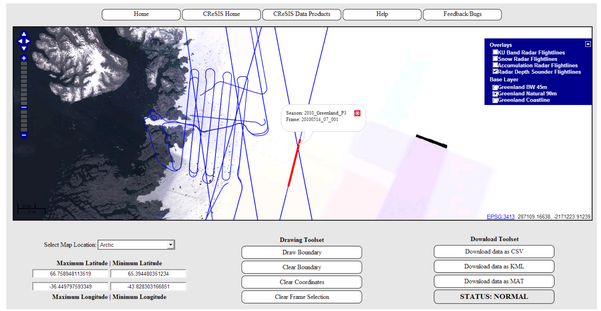
The CReSIS MATLAB data picker was completely overhauled in order to be compatible with the new OPS system. As well as a complete visual overhaul of the GUI the functional background was completely revamped as well. While echograms are currently still loaded from the local file system the layer data, and map data are retrieved from the OPS server. As **layers are tracked** the data is **saved to the DBMS** by the user and becomes immediately available to the web client and all other picker clients connected to the system. The picker allows for a theoretically infinite number of layers to be tracked by implementing a layer manager. In the new system not only surface and bottom can be tracked, but any layer identifiable in the image. Below is the new CReSIS layer picker. Shown is the preference window (top left) which controls the source data and map selection, the map window (bottom left) which displays the available data and allows for text search or geographic selection of data segments, and the echogram "pick" window (right) which is the actual layer tracking interface.





JavaScript Web Client

The web client is the **public interface** to all data stored in the DBMS. Currently the primary purpose of the website is **distribution of data**; however, analysis capabilities will be implemented in the future exposing the power of SQL and PostGIS. A JavaScript **echogram image browser** is also in development. Currently the client is built using OpenLayers, but will be updated with more features using the **GeoEXT JavaScript** library on top of the **OpenLayers** interface. Current features include geographically based data retrieval, data availability visualization, and real-time data updates since the web client is tied directly to the DBMS. CReSIS will also host a primary copy of the web client that will allow access to all of the publicly available data via the convenient JavaScript interface. Below is an image of the first version of the web client.



References & Additional Information

http://www.opengeospatial.org/standards/wms
http://www.opengeospatial.org/standards/wfs
OpenPolarServer logo by Aaron Taveras
https://www.djangoproject.com/
http://www.vagrantup.com/
https://www.virtualbox.org/
http://openlayers.org/
http://geoserver.org/
http://www.geoext.org/
http://tomcat.apache.org/
http://www.centos.org/
http://httpd.apache.org/
http://www.postgresql.org/
http://postgis.net/

CReSIS
Center for Remote Sensing of Ice Sheets

KU RESEARCH & GRADUATE STUDIES The University of Kansas
NSF
NASA
THE KANSAS BOARD OF REGENTS 1925
KU SCHOOL OF ENGINEERING The University of Kansas
KU

This material is based upon work supported by the **National Science Foundation** under Grant No. **ANT-0424589**. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.