## Objective

Create a web-based data service that serves global climate model data at a specific point in space and time for a given set of climate variables.

Gridded climate model output is great if you want to examine the spatial patterns of climate in a single time slice, but is difficult to use if you want to track the trajectory of a specific point in space (x, y) in climate space through time. I know of no way to programmatically and efficiently get the value of climatic variables (maximum temperature, minimum temperature, precipitation, etc) at a single grid cell at a time slice using standard web-based repositories. My goal is to create a data service that allows a client to query a space-time point (e.g., 37N, -122W, 1250 years ago) for a specific climate layer and return the information in a consumable format (json). I’d like whatever I build to be web-based, and have the heavy lifting done on the server side, so that the user doesn’t need to download the whole data file (e.g., netcdf) which would be a lot of data transfer and probably contain a lot of unnecessary information. I’d also like to build out a REST interface so that the querying can be efficiently done by any client. The service would serve downscaled CCSM3 climate model output for North America between 22,000 years ago and the present, as well as, potentially, other climate models and/or other gridded datasets (soils, land use) though both of these seem like they’d be much more complicated.

## Challenges

**Speed**: I’m building a visualization tool, so response times and network latency are important issues to consider. A prototype of the service ran too slowly (and returned too much data) to be efficiently consumed by visualization clients (> 30 s for a moderately sized request). There are two key ways I am planning on improving the speed: (1) database query optimization/spatial table indexing and (2) response compression. Combined I think would provide a reasonable solution.

**Metadata:** Climate models (and other gridded datasets) require a lot of metadata to be correctly interpreted. For example: variable, variable units, climate model, climate model source, spatial resolution, spatial extent, time slice (e.g., years before present), averaging period, temporal resolution. Clearly, there is a lot of information to keep track of. Netcdf file formats store all of this information nicely in their headers and variable descriptions. A database solution needs to efficiently store and access this information in a table structure to see if it matches a user query. Also, a database solution is rigid, so there need to be enough columns to describe any conceivable climate model dataset. The data model is proving tricky to get right.

## Solution

**Database**: Run a Postgres database with PostGIS spatial extensions. Store each time slice (e.g, 22000 years ago, 21900 years ago, 21800 years ago…) for each variable (e.g., January precipitation, January maximum temperature,…) as a separate postgres raster table table. Store a pointer to that table in a central index table that keeps track of the layer’s metdata. When a client issues a query, search the index table to find the row(s) that meet the user’s query. Go to the raster tables that are referred by each of these rows and do a point-in-raster value extraction. Send back the values at those points in those tables.

**Server/Service:** Run a Node.js server interface. Using the Express.js and Postgres (pg) bindings for node, parse client requests and send them to the database. When the database responds, package up the json and send the response back to the client.

## Technical Requirements

**Development Server**: I need a development server on which to implement the solution outlined above. Given the need for a Node/postgres configuration, a linux server or virtual instance would be the best operating system. The server would also need to support Node and a postgres database. Ideally, it would also have git, which would speed up development. The hardware requirements for the server would be very modest. Currently, we have only ~15G of CCSM3 data that would need to go onto the development server and into the database. More layers are possible in the future, but we’ll stick with these for the time being. Memory and CPU requirements are not large, since no calculations will really be done on the server. Moreover, while in development, the traffic will be super low (just me, most likely). We’d need a static IP for the server, but wouldn’t necessary need a public DNS domain listing for it. Eventually, I envision this as a public service that can be consumed independently via its API, but for now, it will just be a data source for Ice Age Mapper.