```
../scripts/design_analysis -p analysis/2B1A_germline.pdb \
-S -b -s nd -c ../input_files/all.corr
../output_files/msd_output_*A*.pdb

>>
#score vs mature
Total Bit Score of Design ===> 50.2318
Total Shannon Entropy of Design ===> 115.9577
Normalized Bit Score for design ===> 0.4337
```

This gives a design score towards 0.35 for the germline sequence and 0.24 for the mature sequence. Everything can be repeated this exact way. For single state design you can keep the fitness file exactly the same, but remove the other states.

```
#initialize the states and what the states file name is
STATE_VECTOR A 2b1a.states
STATE_VECTOR B 2xwt.states
STATE_VECTOR C 3hmx.states


#tell design to minimize energy for each state
SCALAR_EXPRESSION best_A = vmin( A )
SCALAR_EXPRESSION best_B = vmin( B )
SCALAR_EXPRESSION best_C = vmin( C )

#Fitness - design to minimize all energies simultaneously
FITNESS best_A
```

And run the exact same procedures.

VI.5.2 Chapter II - Database and Design

This section accompanies chapter II. I will go over uploading the sequences to a database, selecting the correct sequences, threading, and finally design.

VI.5.2.1 Sequence Analysis

The methods section detailed how I actually sequence the amplicons from 64 healthy donors, but processing them takes quite a bit of computational work. The VANTAGE core at Vanderbilt returns the sequencing runs as two paired end reads in FASTQ format. They must be "stiched" together to make one read. For example, I have provided 100,000 sequences from the forward and reverse reads to be stiched togher from donor 54. I use a stitching algorithm called "pandaseq" to process theses (Bartram et al., 2011). These commands are incredibly simple to run but do require that you first download and install "pandaseq" (https://github.com/neufeld/pandaseq).

```
/usr/local/bin/pandaseq -f input_files/unstiched/
    unstiched_forward.fastq -r input_files/unstiched_reverse.
    fastq -T 23 > stiched/stiched.fasta
```

Pandaseq will automatically output to the fasta format which is convenient for the next step. Please note, that although I have put 100,000 sequences to be stitched together, pandaseq will only find about a quarter of them. That is because I have not provided the entire file. However, I have provided 100,000 sequences in the directory "input_files/stiched/stiched.fasta" to be used in subsequent steps. To analyze the sequences I use PyIg, my own sequence aligner against Ig mAbs that's based on IgBLAST (Ye et al., 2013). I will probably publish this soon when it is more stable. For human IgG's it works incredibly simple to use. For the full source code email jwillis0720@gmail.com, as it's currently unpublished at the moment.

```
./PyIg
usage: igblast [-h] -q query.fasta [-d DB_PATH] [-i
    INTERNAL_DATA]
                [-a AUX_PATH] [-y {Ig,TCR,custom}] [-or {
                    human,mouse}]
                [-nV NUM_V] [-nD NUM_D] [-nJ NUM_J] [-dgm
                    D_GENE_MATCHES]
                [-s {imgt,kabat}] [-x EXECUTABLE] [-o OUT] [-
                    t TMP]
                [-e E_VALUE] [-w WORD_SIZE] [-pm
                    PENALTY_MISMATCH]
                [-nP NUM_PROCS] [-op OUTPUT_OPTIONS] [-z] [-c
                    ] [-j]

optional arguments:
  -h, --help            show this help message and exit

Necessary:
  These have to be included

  -q query.fasta, --query query.fasta
                        The fasta file to be input into
                            igBlast

Database Paths:
  -d DB_PATH, --db_path DB_PATH
                        The database path to the germline
                            repertoire
  -i INTERNAL_DATA, --internal_data INTERNAL_DATA
                        The database path to internal data
                            repertoire
  -a AUX_PATH, --aux_path AUX_PATH
                        The auxilariay path that contains
                            the frame origins of the germline
                             genes for each repertoire.
```

```
                        Helps produce translation and other
                            metrics

IgBlast Sprecific:
  IgBlast Specific Options with a Default

  -y {Ig,TCR,custom}, --type {Ig,TCR,custom}
                        Is this an IG or TCR recombination
  -or {human,mouse}, --organism {human,mouse}
                        The organism repeortire to blast
                            against
  -nV NUM_V, --num_v NUM_V
                        How many V-genes to match?
  -nD NUM_D, --num_d NUM_D
                        How many D-genes to match?
  -nJ NUM_J, --num_j NUM_J
                        How many J-genes to match?
  -dgm D_GENE_MATCHES, --d_gene_matches D_GENE_MATCHES
                        How many nuclodtieds in the D-gene
                            must match to call it a hit
  -s {imgt,kabat}, --domain {imgt,kabat}
                        Which classification system do you
                            want

General Settings:
  -x EXECUTABLE, --executable EXECUTABLE
                        The location of the executable,
                            default is /usr/bin/igblastn
  -o OUT, --out OUT     output file prefix
  -t TMP, --tmp TMP     temporary directory to store files
    in.
                        Defaults to ./tmp
  -e E_VALUE, --e_value E_VALUE
                        Real value for excpectation value
                            threshold in blast.
                        Put in scientific notation
  -w WORD_SIZE, --word_size WORD_SIZE
                        Word size for wordfinder algorithm
  -pm PENALTY_MISMATCH, --penalty_mismatch PENALTY_MISMATCH
                        Penalty for nucleotide mismatch
  -nP NUM_PROCS, --num_procs NUM_PROCS
                        How many do you want to split the
                            job across, default is the number
                             of processors
```

```
Outputting Options:
  -op OUTPUT_OPTIONS, --output_options OUTPUT_OPTIONS
                        Open this file and comment out
                          options you don't want in your
                          final file.
                        The first column is the name of the
                          option.
                        The second column is used by the
                          parser and should not be changed.
  -z, --zip             Zip up all output files
  -c, --concatenate     Turn off automatic concatenation and
      deletion of temporary files.
                        Files are split up at the beginning
                          to run across multiple processors
  -j, --json            Use the JSON output option that will
      format the text driven igblast output to a json
     document.
                        Defaults to CSV
```

I'll keep it simple for the protocol capture, but want to show how robust PyIg can be. Assume you are in the PyIg directory under PyIg/src/

```
python2.7 execution.py -q input_files/stiched/stiched.fasta
   -d datafiles/database/ -i internal_data/ -a datafiles/
   optional_file/ -y Ig -or human -nV 1 -nD 1 -nJ 1 -s imgt
   -o protocol_capture -nP 23 -z -j
```

Each option is listed in the help. Make sure you see which each one does. For instance -nP needs to be carefully considered since it is the number or processors it will be run on. The output to this command line is a protocol_capture.json.gz file that will be uploaded to a Mongo database. One entry (with default output settings) looks like this.


```
{
    "_id" : "donor_10_1000",
    "raw_seq" : "TGGAGCTGAGCAGCCTGAGATCTGAGGACACGGCCGT
    ATATTACTGTGCGAAAGAACTATATGATAGTAGTGGTTATTACTACTTCC
    TGCCTTCTTACTACTACTACGGTATGGACGTCTGGGGCCAAGGGACCACG
    GTCACCGTCTCCTCAGGTAAG",
    "d_region" : "TATGATAGTAGTGGTTATTACTAC",
    "cdr3_aa" : "AKELYDSSGYYYFLPSYYYYGMDV",
    "fw4_aa" : "WGQGTTVTVSSGK",
    "full_seq_aa" : "AKELYDSSGYYYFLPSYYYYGMDVWGQGTTVTVSSGK",
    "cdr3" : "GCGAAAGAACTATATGATAGTAGTGGTTATTACTACTTCCTGCCTT
    CTTACTACTACTACGGTATGGACGTCTG",
    "top_d" : "IGHD3-22*01",
    "v_d_junction" : "ACTA",
    "top_j" : "IGHJ6*02",
```

```
    "cdr3_aa_length" : 24,
    "fw4" : "TGGGGCCAAGGGACCACGGTCACCGTCTCCTCAGGTAAG",
    "d_j_junction" : "TTCCTGCCTTCT",
    "d_or_j_junction" : "",
    "top_v" : "IGHV1-69*06",
    "full_seq" : "GCGAAAGAACTATATGATAGTAGTGGTTATTACTACTTCCT
    GCCTTCTTACTACTACTACGGTATGGACGTCTGTGGGGCCAAGGGACCACGGTCA
    CCGTCTCCTCAGGTAAG",
    "full_seq_aa" : "AKELYDSSGYYYFLPSYYYYGMDVWGQGTTVTVSSGK",
    "d_bit_score" : 48.3,
    "d_evalue" : "3e-10.0",
    "d_alignment_length" : 24,
    "d_query_seq" : "TATGATAGTAGTGGTTATTACTAC",
    "d_subject_seq" : "TATGATAGTAGTGGTTATTACTAC",
    "d_percent_identity" : 100,
    "d_percent_positives" : 100,
    "d_mismatches" : 0,
    "d_positives" : 24,
    "d_identical" : 24,
    "d_subject_length" : 31,
    "d_score" : 24,
}
```


To download and install mongo, consult the manual (http://docs.mongodb.org/manual/installation/). This installation will also include all the tools that making uploading files very easy. The output of PyIg should be "protocol_-capture.json.gz". We can then upload this json file to mongodb using the "mongoimport" application.

Note:
I assume you have an instance of MongoDB running. Please consult the documentation to run an instance of MongoDB. For example,

```
mongod --dbpath mongo_db_data/ &

gunzip analyzed_files/protocol_capture.json.gz &&
    mongoimport -d test_database -c protocol_capture --file
    analyzed_files/protocol_capture.json
```

Here I have made a database called test_database and a collection called protocol_capture. First thing to do is remove redundancies within mongo. To do that, I can make a simple index on the "Input Sequence" key and drop duplicates.

```
mongo test_database
>db.protocol_capture.ensureIndex({'cdr3_aa':1},{unique:true,
     dropDups:true})
```

The next thing I want to do is remove non-productive sequences. PyIg outputs a productive field, using mongo, I can tell the database to drop any "document" that contains a stop codon in the HCDR3.

```
>db.protocol_capture.remove({"productive_cdr3":"False"})
```

For ROSETTA, I want only the thirty length HCDR3s. I can use "mongoexport" for this query along with an 'awk' statement to get out the 30 length fasta files.

```
mongoexport -d test_database -c protocol_capture -q '{"
    cdr3_aa_length":30}' --csv -fields "_id","cdr3_aa" | awk
    -F "," '{gsub("\"","",$1);gsub("\"","", $2);print(">"$1"\
    n"$2)}' > 30_length.fasta
```

This fasta file will be used in the remainder of the protocol. I will not go over all the different types of analysis I can do with this database for the purpose of this protocol.

VI.5.2.2 PSSM Heuristics

Using the fasta file "analysed_files/30_length.fasta" generated in the previous section, a quick script written in python to get random 2,000 sequences is shown below.

```
#!/usr/bin/env python
from Bio import SeqIO
import random
import sys
handle = open(sys.argv[1])
records = SeqIO.parse(handle, "fasta")
dictionary = {}
for seq in records:
    dictionary[seq.id] = str(seq.seq)
random_dict = random.sample(dictionary.items(),2000)

with open(sys.argv[2],'w') as f:
     for seq in random_dict:
          f.write(''>{0}\n{1}\n''.format(seq[0],seq[1]))
```

And use it to get 2000 random sequences:

```
./get_2000_random.py  ../analyzed_sequences/30_length.fasta
    2000_random.fasta
```

Using ROSETTADESIGN, I will generate 2,000 resfiles that tell the packer to mutate the HCDR3 into each of the entries in the fasta file. To do this, there is a script available from the Meiler lab called "fasta_to_resfile.py" which will generate the resfiles necessary.

```
fasta_into_res.py 2000_random.fasta 95 126 H 0
```

The rest of the protocol uses ROSETTASCRIPTS to do the design. For ROSETTASCRIPTS, you need an xml file, options file, and command line. The xml file is a scripting file that

tells ROSETTA specific set of instructions (here I will name it threading.xml). The first step is relatively simple only doing a design and a full-atom minimization (called relax).

```
<dock_design>
     <SCOREFXNS>
     </SCOREFXNS>
     <FILTERS>
     </FILTERS>
     <TASKOPERATIONS>
      <InitializeFromCommandline name=ifcl/>
      <ReadResfile name=rr filename=%%resfiles%% />
     </TASKOPERATIONS>
          <MOVERS>
           <PackRotamersMover name=pr task_operations=ifcl,rr/
              >
           <FastRelax name=fr task_operations=ifcl/>
          </MOVERS>
          <APPLY_TO_POSE>
          </APPLY_TO_POSE>
          <PROTOCOLS>
           <Add mover_name=pr/>
           <Add mover_name=fr/>
          </PROTOCOLS>
</dock_design>
```

The only caveat here is that we specify the resfile as a variable so the protocol does not have to be hard-coded. The command line will specify each resfile to give to the job. First, an option file must be produced as a simple text file.

```
-out
    -pdb_gz
-parser
    -protocol input_file/threading.xml
-s input_files/input_pg9_no_antigen.pdb
-nstruct 100
```

Lastly the command line which will include the resfile as an option (named run.csh).

```
#!/bin/csh
set res = $1
set out = `basename $res .resfile`
mpiexec -n 101 my/rosetta/pathrosetta_scripts.mpistatic.
   linuxgccrelease @options.txt -out:prefix $out -parser:
   script_vars resfiles=${res} > out.log
```

And to run the command I simply use an awk script to generate all the commands.

```
ls *resfile | awk '{system( "run.csh "$1 ) }'
```

Note: This should be run on a computer cluster as 100 processors are needed in the above protocol
The next step is to run the output PDB files and generate a position-specific scoring matrix. This is easily accomplished with the "create_pssm_from_threading.py" script that is also found in the Meiler lab scripts repository. A simple command to generate a PSSM.

```
./create_pssm_from_threading.py -g -r resfiles PG9.resfile -
    n 2000.p3sm *.pdbs
```

The output .p3sm can now be used to predict the top N sequences from the 30_length.fasta generated earlier in the protocol.

```
./create_pssm_from_threading.py -r resfiles/PG9.resfile -s -
    p 2000.p3sm analysis_files/30_length.fasta
```

This generates a file called "scored_fasta.output". I use awk and some other gnu commands to get the top 1000 scored fasta files.

```
sort -nk 3 scored_fasta.output | head -n 1000 | awk '{print
    (">"$1"\n"$2)}' > top1000.fasta
```

Finally, I can make all the resfiles using the same command as before.

```
fasta_into_res.py top1000.fasta 95 125 H 0
```

For the full design protocol using these sequences and resfiles. See the next section on PG9 redesign (subsection VI.5.3).

VI.5.3 Chapter III - PG9 Design

This protocol capture will detail the how to use ROSETTADESIGN to predict mutations that enhance specificity. This accompanies the manuscript Willis *et al.* ***Nature Med.*** (submitted). It assumes that you have a ROSETTA license from www.rosettacommons.org.

VI.5.3.1 Preparing the input files

Using PG9/CAP45 complex, I have prepared a ROSETTA compatible file called PG9_input.pdb. This has spcecial identifiers for the glycans that ROSETTA's database can understand. To create your own glycan input, an excellent protocol capture is provided in an accompanying manuscript by Doug Renfrew (Renfrew et al., 2012).

The design protocol used runs through the following steps.

- Favor native residue - gives bonuses to sequences which match PG9*wt*
- Design/minimize/dock iteratively
- Constrain movements so glycans retain input position
- Relax the energy of the structure
- Re-dock