```
#include <iostream>
#include <fstream>
#include <cmath>
#include <valarray>
#include <string>
#include <sstream>
#include "bGrid.h"
#include <iomanip>
#include <string.h>
using namespace std;

const unsigned int DEBUG_COUNT=2;

/* Templates */
template< typename T, std::size_t N > inline
std::size_t size( T(&)[N] ) { return N ; }

/*
  Constructors and Destructors
*/
bGrid::bGrid() : isInRes_(0), residueCount_(0), res_(0.5), fit_(2), thk_(0)  {
  for(unsigned int i = 0; i < size(min_); ++i) {
    min_[i] = 1000.0;
    max_[i] = -1000.0;
  }
}
bGrid::~bGrid() {}

/*
  Read in Protein from File
*/
bool bGrid::readPoints(char *file) {
  saveFilename(file); // save the filename (no extension)
  protPnt_ = _readPoints(file,prt_,1); // save the protein points
  return protPnt_;
}
bool bGrid::readPoints(char *file, char *file2) {
  saveFilename(file); // save the filename (no extension)
  protPnt_ = _readPoints(file,prt_,1); // save the protein points
  centPnt_ = _readPoints(file2,cnt_,0); // save the centroid points
  return (protPnt_ & centPnt_);
}
bool bGrid::_readPoints(char *file, std::valarray<float> &pnts, bool save) {

  // open the file
  ifstream ip;
  ip.open(file, ifstream::in);
  if(!ip) {
    return 0;
  }
  printf("File: %s\n",file);

  // read in the file... not the most efficient, bleh
  // -- we read in w/o doing anything to get a cound of the residues
  vector<string> data;
  string bffr;
  for(int i = 0; getline(ip,bffr); ++i) {
    data.push_back(bffr);
  }
  ip.close();

  // save how many points we have
  if(save) {
    residueCount_ = data.size();
```

```
  }

  // resize the array and prepare the stream
  // -- this is an unfortunate side effect of using the valarray
  // => it clears everything upon resize =/
  pnts.resize(data.size()*3);

  // loop through each line and save the coordinates
  for(unsigned int i = 0; i < data.size(); ++i) {
    istringstream ss(data[i]);
    float flt;

    // loop through each value on the line
    for(int j = 0; ss >> flt; ++j) {
      pnts[(i*3) + j] = flt;
    }
  }

  return 1;
}

/* Find min and max */
void bGrid::findMinMax() {
  // Loop over each axis and calculate the min and max values
  int numPnts = prt_.size()/3;
  for(int i = 0; i<3; ++i) {
    min_[i] = (int)valarray<float>(prt_[slice(i,numPnts,3)]).min();
    max_[i] = (int)valarray<float>(prt_[slice(i,numPnts,3)]).max();
    cout << min_[i] << " : " << max_[i] << endl;
  }
}

/*
  Recenter protein completely within positive boundaries
*/
bool bGrid::repositionPoints() {
  if(protPnt_) {
    // find max and min...and a few other things
    int numPnts = prt_.size()/3;
    for(int i=0; i<3; ++i) {
      // 1) find the min and max
      // 2) adjust them to account for desired grid spacing
      //   -- i.e. go from protein point min and max to grid point
      min_[i] = (int)valarray<float>(prt_[slice(i,numPnts,3)]).min() - (fit_ + thk_);
      max_[i] = (int)valarray<float>(prt_[slice(i,numPnts,3)]).max() + (fit_ + thk_);
      cout << min_[i] << " : " << max_[i] << endl;

      // 3) save the correction amount -- still in angstroms
      correction_[i] = min_[i];

      // 4) reposition the min and max to positive numbers
      max_[i] -= min_[i];
      min_[i] = 0;
      cout << min_[i] << " : " << max_[i] << endl;
    }

    // adjust the protein points
    this->protPnt_ = this->_repositionPoints(this->prt_);
  }
  if(centPnt_) {
    this->centPnt_ = this->_repositionPoints(this->cnt_);
  }
  return (this->protPnt_ & this->centPnt_);
}
```

```
bool bGrid::_repositionPoints(std::valarray<float> &pnts) {
  int numPnts = pnts.size() / 3; // get number of points
  for(int i=0; i<3; ++i) {
    slice s = slice(i,numPnts,3); // create a slice -- single axis
    pnts[s] = valarray<float>(pnts[s]) - correction_[i]; // adjust protein
    cout << i << ": " << correction_[i] << ", " << pnts[i] << endl;
  }
  return 1;
}



/* Initialize Grid */
void bGrid::initializeGrid() {
  // Check and see if we need extra space, i.e. the z-axis is too big
  // -- if so, double the grid size
  int addGridSpace = this->checkRes();
  cout << addGridSpace << endl;

  // adjust everything to resolution space
  this->setToResSpace();

  for(int i=0; i<3; ++i) {
    cout << min_[i] << " : " << max_[i] << endl;
  }

  // initialize grid
  // -- the min should be 0 on all accounts
  // -- add 1 since max is the last index, not the size
  int grdDimensions = addGridSpace * (max_[0]+1) * (max_[1] + 1);
  grd_.resize(grdDimensions, 0x0);
  tmp_.resize(grdDimensions, 0x0);
  cout << grdDimensions << endl;

  return;
}

/* Check required resolution */
int bGrid::checkRes() {
  numRes_ = 1;
  int resCheck = (max_[2] / res_);
  while(resCheck > 63) {
    resCheck -= 63 - ((fit_ + thk_) / res_);
    highRes_ = true;
    ++numRes_;
  }
  return numRes_;
}

/* set everything to resolution space */
bool bGrid::setToResSpace() {
  if(isInRes_) {
    return 0;
  }

  prt_ /= res_; // protein points
  cnt_ /= res_;
  fit_ /= res_; // paramters
  thk_ /= res_;
  for(int i=0; i<3; ++i) { // min and max
    min_[i] /= res_;
    max_[i] /= res_;
  }
```

```
  isInRes_ = true;
  return 1;
}

/* set everything to normal space */
bool bGrid::setToNormalSpace() {
  if(!isInRes_) {
    return 0;
  }

  prt_ *= res_; // protein points
  cnt_ *= res_;
  fit_ *= res_; // parameters
  thk_ *= res_;
  for(int i=0; i<3; ++i) { // min and max
    min_[i] *= res_;
    max_[i] *= res_;
  }

  isInRes_ = false;
  return 1;
}

/*
  Initialize Exclusion and Inclusion Matrices (or grids...yes, I know)
*/
bool bGrid::initializeStamps() {
  int radius = fit_; // exclusion
  haveExc_ = this->_initializeStamp(radius,exc_,exsl_);

  radius = thk_ + fit_; // inclusion
  haveInc_ = this->_initializeStamp(radius,inc_,insl_);

  printStamp();
  return 1;
}

/*
  Initialize Stamp
*/
bool bGrid::_initializeStamp(int radius, std::valarray<unsigned long> &stamp, std::vector< valarray<size_t> > &stampSlice) {
  // radius from calling function: int radius = (fit_ / res_);
  int diamet = 2 * radius;

  // initialize bins
  int binCnt = 0;
  for(int i = 0; i <= (radius); ++i) {
    binCnt += i;
  }

  // resize! (and fill in with ones)
  stamp.resize(binCnt, 0);//(pow(2,diamet) - 1));
  stampSlice.resize(binCnt);

  // loop through each outer layer of the grid
  int index = 0;
  for(int k = 0; k < (radius); ++k) {

    // loop through each unique position on the outer layer
    for(int i = k; i < (radius); ++i) {

      // initialize the valarray
      stampSlice[index].resize(8);
```

```
        int slc = 0;

        // calculate each of the eight points
        // note: these are mapped to the grid, not to the small box of the
        // exclusion. MEANING: we only need to add the x dimension (usually i
        // in our implementation) to the indirect arrays to get the
        // appropriate index on the grid.
        int n = max_[0]+1;
        int p = 2*radius - 1;
        stampSlice[index][slc] = i + (k*n);
        stampSlice[index][++slc] = (p-i) + (k*n);
        stampSlice[index][++slc] = (i*n) + k;
        stampSlice[index][++slc] = (i*n) + (p-k);
        stampSlice[index][++slc] = (p-i)*n + k;
        stampSlice[index][++slc] = (p-i)*n + (p-k);
        stampSlice[index][++slc] = (p-k)*n + i;
        stampSlice[index][++slc] = (p-k)*n + (p-i);

        ++index;
      }
    }

    // setup the middle point
    valarray<float> mid( (((float)diamet -1)/2) , 3);

    // go through each point in the exclusion matrix...for the unique bins
    index = 0;
    float scaledLength = radius - res_;
    for(int k = 0; k < radius; ++k) {
      for(int i = k; i < radius; ++i) {
        for(int z = 0; z < radius; ++z) {
          // create temp valarray for the point
          float tmp[] = {(float)k, (float)i, (float)z};
          valarray<float> gpt(tmp,3);

          // check the distance between mid pt and tmp pt
          // -- if w/in range, add symmetrically
          if(pointDistance(mid,gpt) < scaledLength) {
            stamp[index] |= (unsigned long)pow(2.0,z);
            stamp[index] |= (unsigned long)pow(2.0,(diamet-1-z));
          }
        }
        ++index;
      }
    }

    return 1;
}

/*
   Calculate Point Distances
*/
float bGrid::pointDistance(valarray<float> &a, valarray<float> &b) {
    valarray<float> c = a-b;
    c *= c;
    return sqrt( c.sum() );
}

/*
   Stamp Protein Points with [In|Ex]clusion Grids
*/
bool bGrid::stampPoints() {
    //~ for(unsigned int i=0; i< DEBUG_COUNT; ++i) {
      //~ int index = i*3;
```

```
    //~ cout << "P: ";
    //~ cout << prt_[index] << " : ";
    //~ cout << prt_[++index] << " : ";
    //~ cout << prt_[++index] << endl;

    //~ cout << "C: ";
    //~ cout << cnt_[index] << " : ";
    //~ cout << cnt_[++index] << " : ";
    //~ cout << cnt_[++index] << endl;
    //~ cout << endl;
  //~ }

  if(haveExc_) {
    int correction = fit_;
    cout << "Stamping Protein" << endl;
    stamped_ = this->_stampPoints(correction,prt_,exc_,exsl_,grd_);
    if(centPnt_) {
      cout << "Stamping Centroids" << endl;
      stamped_ = this->_stampPoints(correction,cnt_,exc_,exsl_,grd_);
    }
  }
  if(haveInc_) {
    int correction = thk_ + fit_ - 1;
    cout << "Stamping Inclusions" << endl;
    //~ stamped_ = this->_stampPoints(correction,prt_,inc_,insl_,tmp_);
  }
  // Reconcile the inclusion and exclusion
  grd_ = grd_ ^ tmp_;

  print2dProtein();


  return stamped_;
}

/*
  Stamp Grid Around a Point
*/
bool bGrid::_stampPoints(int correction, std::valarray<float> &pnts,std::valarray<unsigned long> &stamp, std::vector< valarray<s

  // Corection: given from calling function
  //~ int correction = (fit_/res_) - 1;

  // displacement and depth of exclusion grid
  int position = 0;
  int depth = 0;

  // Resize the protein to the proper resolution
  // -- we initialized the grid to the resolution
  //~ pnts /= res_;

  // shift point from the center box to the corner
  pnts -= correction;

  // create slices
  slice xs = slice(0,residueCount_,3);
  slice ys = slice(1,residueCount_,3);
  slice zs = slice(2,residueCount_,3);

  int allowableDepth = 63;
  if(highRes_) {
    allowableDepth = 63 - (fit_ + thk_);
  }
```

```
// Loop through protein points
for(unsigned int i=0; i<DEBUG_COUNT;++i) { //pnts.size()/3; ++i) { // DEBUG_COUNT;++i) { //

  // a little ugly due to valarray implementation
  int index = i*3;
  int x = pnts[index];
  int y = pnts[++index];
  int z = pnts[++index];

  cout << "STAMPING:" << endl;
  cout << "\t" << x << endl;
  cout << "\t" << y << endl;
  cout << "\t" << z << endl;

  // save old positions
  int displacement = position;
  int perspective = depth;

  // calculate new positions
  position = ((y)*(max_[0]+1) + (x));
  depth = z;

  // check and adjust for high resolution
  if(z > allowableDepth) {
    adjustPlacement(depth,position);

    cout << "IN EXTRA!!!" << endl;
    cout << "\t" << position << endl;
  }

  // adjust according to previous placement
  displacement = position - displacement;
  perspective = depth - perspective;

  cout << "Position: " << endl;
  cout << "\t" << position << endl;
  cout << "\t" << displacement << endl;

  cout << "Depth: " << endl;
  cout << "\t" << depth << endl;
  cout << "\t" << perspective << endl;

  // Loop through exclusion slice, displace each, and EXCLUDE
  // -- might it be faster to simply multiply rather than test?
  for(unsigned int g=0; g<stamp.size(); ++g) {

    stampSlice[g] += displacement;
    stamp[g] *= pow(2.0,perspective);

    //~ if(z>allowableDepth) {
      //~ cout << "Grid:" << endl;
      //~ cout << "\t" << stampSlice[g][0] << endl;
      //~ cout << "\t" << stamp[g] << endl;
      //~ cout << "\t" << paper[stampSlice[g][0]];
      //~ cout << endl;
    //~ }
      //~ cout << "\t";
      //~ printSingleHex(cout,stamp[g],63);
      //~ cout << "\t";
      //~ printSingleHex(cout,paper[stampSlice[g][0]],63);

    // EXCLUDE POINTS!!
    paper[stampSlice[g]] = stamp[g] | valarray<unsigned long>(paper[stampSlice[g]]);
```

```
        //~ if(z>allowableDepth) {
          //~ cout << "\t";
          //~ printSingleHex(cout,paper[stampSlice[g][0]],63);
        //~ cout << endl;
        //~ }
      }  // end loop through exclusion space
      cout << "END" << endl;

      //~ cout << endl;

    } // end loop through points

    // Reset protein to original location
    pnts +=  correction;
    //~ pnts *= res_;
         cout << "END" << endl;
    // Reset stamp to original location
    stamp /= pow(2.0,depth);
         cout << "END" << endl;
    for(unsigned int i=0; i<stampSlice.size(); ++ i) {
      stampSlice[i] -= position;
    }
         cout << "END" << endl;
    return 1;
}

bool bGrid::adjustPlacement(int &depth, int &position) {
    int tooBig = 63 - (fit_ + thk_);
    int howBig = 1;
    while(depth > tooBig) {
      depth -= tooBig;
      ++howBig;
      if(howBig == numRes_) {
        tooBig = 63;
      }
    }
    position += (max_[0] + 1) * (max_[1] + 1);
    return 1;
}


/*
   Find Nearby Points
*/
int bGrid::countNearby(std::valarray<float>&a) {
    // adjust coordinates from center to corner
    a -= fit_ - 1;

    // initialize placement coordinates
    int position = (a[1] * max_[0]) + a[0];
    int depth = a[2];

    // check and adjust for high resolution
    int allowableDepth = 63 - (fit_ + thk_);
    if(depth>allowableDepth && highRes_) {
      depth -= allowableDepth;
      position += (max_[0] + 1) * (max_[1] + 1);
    }

    // initialize counting variables
    int nearby = 0;
    std::valarray<unsigned long> overlap(exc_.size());

    //~ cout << "STAMPING:" << endl;
```

```
    //~ cout << "\t" << a[0] << endl;
    //~ cout << "\t" << a[1] << endl;
    //~ cout << "\t" << a[2] << endl;

    //~ cout << "Position: " << endl;
    //~ cout << "\t" << position << endl;

    //~ cout << "Depth: " << endl;
    //~ cout << "\t" << a[2] << endl;

    for(unsigned int i=0; i<exsl_.size(); ++i) {
      exsl_[i] += position;
      exc_[i] *= pow(2.0,depth);

      //~ cout << endl;
      //~ cout << i << endl;

      //~ cout << "Grid:" << endl;
      //~ cout << "\t" << exc_[i] << endl;
      //~ cout << "\t" << exsl_[i][0] << endl;
      //~ cout << "\t" << grd_[exsl_[i][0]];
      //~ cout << endl;

      //~ cout << "\t";
      //~ printSingleHex(cout,exc_[i],63);
      //~ cout << "\t";
      //~ printSingleHex(cout,grd_[exsl_[i][0]],63);
      //~ cout << "\t";
      //~ overlap[i] = exc_[i] & grd_[exsl_[i][0]];
      //~ printSingleHex(cout,overlap[i],63);

      overlap[i] = exc_[i] & grd_[exsl_[i][0]];
      overlap[i] >>= int(depth - fit_);
      //~ cout << "Hamming: " << overlap[i] << " : " << hamming_[overlap[i]] << endl;

      nearby += hamming_[overlap[i]];

      exc_[i] >>= (int)depth;
      exsl_[i] -= position;
    }

    // return coordinates to corner
    a += fit_ - 1;

    return nearby;

  /* Hamming table
    If we can store a lookup table of the hamming function of every 16 bit integer, we can do the following to compute the Hamming w
      static unsigned char wordbits[65536] = { bitcounts of ints between 0 and 65535 };
      static int popcount( unsigned int i )
      {
          return( wordbits[i&0xFFFF] + wordbits[i>>16] );
      }
  */

  }

  /*
    Setup Hamming Table
  */
  void bGrid::setupHammingTable() {
    int a = pow(2,16);
    for(int i =0; i<a; ++i) {
      hamming_[i] = this->calculateHammingWeight(i);
```

```
  }
  return;
}

/*
  Calculate Hamming Weight
*/
int bGrid::calculateHammingWeight(int x) {
  int i=0;
  for(i; x; ++i) {
    x &= x-1;
  }
  return i;
}

/*
  Get Hamming Weight for 64 bit numbers
*/
unsigned long bGrid::getHammingWeight(unsigned long x) {
  int weight = 0;
  for(int i=0; i<4 && x > 0; ++i) {
    //~ unsigned long b = 0xFFFF;
    //~ printSingleHex(cout,x,64);
    //~ printSingleHex(cout,b,64);
    weight += hamming_[int(x&0xFFFF)];
    //~ cout << "\tAND: " << (x&0xFFFF) << endl;
    //~ cout << "\tWEIGHT: " << hamming_[int(x&0xFFFF)] << endl;
    x >>= 16;
  }
  return weight;
}

/*
  Print Grid

  ** Removed print 3d exclusion -- wasn't working correctly. Rewrite!
*/
void bGrid::print3dStamp(ostream &out,int radius, valarray<unsigned long> &stamp,std::vector< valarray<size_t> > &stampSlice)

  return;
}
void bGrid::print3dGrid(ostream &out, valarray<unsigned long> &grid, int x) { this->print3dGrid(out,grid,x,x); }
void bGrid::print3dGrid(ostream &out, valarray<unsigned long> &grid, int x, int d) {
  for(unsigned int k=0; k < (grid.size()/x); ++k) {
    for(int i=0; i < x; ++i) {
      for(int z = 0; z < d; ++z) {
        unsigned long a = pow(2.0,z);

        switch(a & grid[k*x + i]) {
          case 0:  out << " " << 0;
            break;
          default: out << " " << 1;
            break;
        }
      }
      cout << endl;
    }
    cout << endl;
  }
}
void bGrid::printSingleHex(ostream &out,unsigned long &one,int z) {
  for(int i=0; i<z; ++i) {
    unsigned long a = pow(2.0,i);
    if(a & one) {
```

```
      out << 1;
    }
    else {
      out << 0;
    }
  }
  out << endl;
}

/*
  Create PyMol Python Script
*/
void bGrid::printPyMol() {

  this->setToNormalSpace();

  // creat the filename
  char *spdb = (char*)malloc(sizeof(char)*64);
  char *sprt = (char*)malloc(sizeof(char)*64);
  strcpy(spdb,prtFile_);
  strcpy(sprt,prtFile_);
  strcat(spdb,".pdb");
  strcat(sprt,"_grid.py");
  strcat(prtFile_,"_grid.py");
  cout << "Writing to: " << sprt << endl;

  // open the file and check
  FILE *op = fopen(sprt, "w");
  if(!op) {
    cout << sprt << " did not open!\n" << endl;
  }

  // Reset to original coordinates
  for(int i=0; i<3; ++i) {
    slice s = slice(i,prt_.size()/3,3);
    prt_[s] = valarray<float>(prt_[s]) + correction_[i];
    s = slice(i,cnt_.size()/3,3);
    cnt_[s] = valarray<float>(cnt_[s]) + correction_[i];
  }

  // Write header
  fprintf(op,"from pymol import cmd\n");
  fprintf(op,"from pymol.cgo import *\n");
  fprintf(op,"\n");
  fprintf(op,"cmd.hide(\"everything\")\n");
  fprintf(op,"cmd.show(\"cartoon\")\n");
  fprintf(op,"cmd.set('transparency','1.0')\n");
  fprintf(op,"cmd.bg_color('white')\n");
  //~ fprintf(op,"FREEMOL=/usr/share/pymol/freemol-trunk/freemol\n");
  //~ fprintf(op,"export FREEMOL");
  fprintf(op,"\n");

  // Write pdb
  //~ fprintf(op,"cmd.load(\"%s\")\n",spdb);
  //~ fprintf(op,"cmd.color(\"firebrick\")\n");
  //~ fprintf(op,"cmd.show(\"stick\")\n");

  // Write origin
  float mi[3];
  float ma[3];
  for(int i=0; i<3; ++i) {
    mi[i] = min_[i] + correction_[i];
    ma[i] = max_[i] + correction_[i];
  }
```

```
fprintf(op,"orgn = [\n");
fprintf(op,"\tCOLOR, 1.0, 1.0, 1.0,\n");
fprintf(op,"\tSPHERE, %.2f, %.2f, %.2f, %.2f,\n",mi[0],mi[1],mi[2], 1.0);
fprintf(op,"\tSPHERE, %.2f, %.2f, %.2f, %.2f,\n",mi[0],mi[1],ma[2], 1.0);
fprintf(op,"\tSPHERE, %.2f, %.2f, %.2f, %.2f,\n",mi[0],ma[1],mi[2], 0.5);
fprintf(op,"\tSPHERE, %.2f, %.2f, %.2f, %.2f,\n",ma[0],mi[1],mi[2], 0.5);
fprintf(op,"\tSPHERE, %.2f, %.2f, %.2f, %.2f,\n",ma[0],ma[1],mi[2], 0.5);
fprintf(op,"\t]\n");
fprintf(op,"cmd.load_cgo(orgn,'orgn')\n");

// Write protein points
fprintf(op,"protein = [\n");
fprintf(op,"\tCOLOR, 0.0, 1.0, 0.0,\n");
for(unsigned int i=0; i<DEBUG_COUNT;++i) { //prt_.size()/3; ++i) {
  int index = i*3;
  fprintf(op,"\tSPHERE, %.2f, %.2f, %.2f, %.2f,\n",
    prt_[index], prt_[index+1], prt_[index+2], 0.2);
}
fprintf(op,"\t]\n");
fprintf(op,"cmd.load_cgo(protein,'protein')\n");

// Write centroid points
if(centPnt_) {
  fprintf(op,"centroid = [\n");
  fprintf(op,"\tCOLOR, 1.0, 0.5, 0.5,\n");
  for(unsigned int i=0; i<DEBUG_COUNT;++i) { //cnt_.size()/3; ++i) {
    int index = i*3;
    fprintf(op,"\tSPHERE, %.2f, %.2f, %.2f, %.2f,\n",
      cnt_[index], cnt_[index+1], cnt_[index+2], 0.1);
  }
  fprintf(op,"\t]\n");
  fprintf(op,"cmd.load_cgo(centroid,'centroid')\n");
}
// Adjust grid points from resolution to angstroms
// Wait...bad idea...good chance we've already
// filled up all available bits...
// -- better do this individually =/
//   grd_ *= res_;

// Write grid points
int allowableDepth = 63;
if(highRes_) {
  allowableDepth -= (fit_ + thk_)/res_;
}
fprintf(op,"grid = [\n");
fprintf(op,"\tCOLOR, 0.0, 0.0, 1.0,\n");
for(int k=0; k<=(max_[1]/res_); ++k) {
  for(int i=0; i<=(max_[0]/res_); ++i) {
    for(int z=0; z<=(max_[2]/res_); ++z) {
      int index = k*((max_[0]/res_)+1) + i;
      // adjust index and depth for highRes
      int depth = z;
      if(z > allowableDepth) {
        adjustPlacement(depth,index);
        //~ index += (max_[0]) * (max_[1]) / res_;
        //~ depth -= allowableDepth;

        //~ cout << "Printing other" << endl;
        //~ cout << "\t" << index << endl;
      }

      if((grd_[index] & (unsigned long)pow(2.0,depth))) {
        fprintf(op,"\tSPHERE, %f, %f, %f, %.2f,\n",
          //~ ((i*res_)+correction_[0]), ((k*res_)+correction_[1]), ((z*res_)+correction_[2]), 0.1);
```

```
                ((i*res_)+correction_[0]), ((k*res_)+correction_[1]), ((z*res_)+correction_[2]), 0.1);
            }
          }
        }
      }
      fprintf(op,"\t]\n");
      fprintf(op,"cmd.load_cgo(grid,'grid')\n\n");
      fprintf(op,"cmd.center('grid')\n");

      // Movie -- Rotate!
      // -- set up the frames
      fprintf(op,"cmd.mclear()\n");
      fprintf(op,"cmd.mset(\"1 x360\")\n");

      // -- simple 2-axis rotation
      fprintf(op,"for i in range(1,120):\n");
      fprintf(op,"\tcmd.mdo(i,'turn x,0.5; turn y,0.5; turn z,0.5')\n");

      // -- rotate and zoom in
      fprintf(op,"for i in range(120,240):\n");
      fprintf(op,"\tcmd.mdo(i,'turn x,0.5; turn y,0.5; turn z,0.5; move z,0.9')\n");

      // -- rotate and zoom out
      fprintf(op,"for i in range(240,360):\n");
      fprintf(op,"\tcmd.mdo(i,'turn x,0.5; turn y,0.5; turn z,0.5; move z,-0.9')\n");
      //~ fprintf(op,"cmd.mplay()\n");

      // Attempts to speed things up
      //fprintf(op,"cmd.set('orthoscopic','on')\n");
      //fprintf(op,"cmd.set('depth_cue','1')\n");
      //fprintf(op,"select cgo01, (all) and not ( (all) within 8 of cgo01) )");
      //fprintf(op,"hide everything, cgo01");
      //fprintf(op,"cmd.ray()");

      // Close the file handle
      fclose(op);

      // Run the command
      // ('cuz windows sucks...)
      //system("pymolwin -r \"D:\\My Dropbox\\grid\\1AWQ_alone_grid.py\"");
    }

    void bGrid::saveFilename(char* file) {
      prtFile_ = (char*)malloc(sizeof(char)*64);
      strncpy(prtFile_,file,strrchr(file,'.')-file);
      return;
    }

    void bGrid::printProtein() {
      // Easy loop through grid -- SAVE
      for(int k=0; k<=max_[1]; ++k) {
        for(int i=0; i<=max_[0]; ++i) {
          int index = k*max_[0] + i;
          for(int z=0; z<=max_[2]; ++z) {
            bool t =  grd_[index] & (unsigned long)pow(2.0,z);
            cout << t << " ";
          }
          cout << endl;
        }
        cout << endl;
      }
    }

    void bGrid::printStamp() {
```

```
  int index = 0;
  for(int i=0; i<fit_; ++i) {
    for(int j=0; j<i; ++j) {
      cout << "  ";
    }
    for(int j=i; j<fit_; ++j) {
      if(exc_[index] > 0) {
        //~ cout << exc_[i] << endl;
        cout << "1 ";
      }
      else {
        cout << "0 ";
      }
      ++index;
    }
    cout << endl;
  }
  cout << exc_.size() << " : " << index << endl;
}

void bGrid::print2dProtein() {
  //~ for(int k=0; k<(max_[1]+1); ++k) {
    //~ printf("%2d: ",k);
    //~ for(int i=0; i<(max_[0]+1); ++i) {
      //~ int index = k*(max_[0]) + i;
      //~ printf("%4d ",i);
    //~ }
    //~ printf("\t:%d\n",max_[0]);
  //~ }
  //~ printf("::%d\n\n",max_[1]);

  //~ for(int k=0; k<(max_[1]+1); ++k) {
    //~ printf("%2d: ",k);
    //~ for(int i=0; i<(max_[0]+1); ++i) {
      //~ int index = k*(max_[0]+1) + i;
      //~ printf("%4d ",index);
    //~ }
    //~ printf("\t:%d\n",max_[0]);
  //~ }
  //~ printf("::%d\n\n",max_[1]);

  for(int k=0; k<(max_[1]+1); ++k) {
    printf("%2d: ",k);
    for(int i=0; i<(max_[0]+1); ++i) {
      int index = k*(max_[0]+1) + i;
      if(grd_[index] > 1) {
        printf("%d ",1);
      }
      else {
        printf("%d ",0);
      }
    }
    printf("\t:%d\n",max_[0]);
  }
  printf("::%d\n\n",max_[1]);

  printf("Correction");
  for(int i=0; i<3; ++i) {
    printf(": %f",correction_[i]);
  }
  printf("\n\n");
}
```