```
#include <iostream>
#include <iomanip>
#include <math.h>
#include <stdlib.h>
#include <fstream>
#include <ctime>
#include <complex>
#include <cstdio>
#include <string>
#include "../MASTER-FILES/AtomParams.h"
using namespace std;


int countGrapheneAtoms (struct vect a1, struct vect a2, struct vect a3, struct vect b1, struct
vect b2, struct vect b3)
{
  struct vect B1, B2, B12, C1, C2, C12;

    // Define TSB super-lattice vectors
  B1 = 4.0*b1;
  B2 = 4.0*b2;
  B12 = B1 + B2;

    // Define boundaries of TSB super-lattice vectors (purely for calculation)
  C1 = B1 + 2.0*b3;
  C2 = B2 + 2.0*b3;
  C12 = B12 + 2.0*b3;

  double slopeC1 = (C1.y - 2.0*b3.y) / (C1.x - 2.0*b3.x);
  double slopeC2 = (C2.y - 2.0*b3.y) / (C2.x - 2.0*b3.x);
  double yInterceptC1 = C1.y - slopeC1*C1.x;
  double yInterceptC2 = C2.y - slopeC2*C2.x;

  double yIntercept1C12 = C12.y - slopeC1*C12.x;
  double yIntercept2C12 = C12.y - slopeC2*C12.x;

    // Initialize graphene basis vectors, where "1" and "2" denote basis atoms of the graphene
    unit cell
  struct vect positionG1 = 2.0*b3;
  struct vect positionG2 = 2.0*b3;

    // Initialize a sufficiently large grid to scan for atoms inside the boundaries
  int maxUnitCells1 = 500;
  int maxUnitCells2 = 500;
  int nAtomsPerLayer = 0;

    // Calculate atomic positions, counting nAtomsPerLayer
  for (int i=-maxUnitCells1; i<maxUnitCells1; i++)
  {
    for (int j=-maxUnitCells2; j<maxUnitCells2; j++)
    {
      positionG1 = i*a1 + j*a2 + a3;
      positionG2 = i*a1 + j*a2 + 2.0*a3;
```

```
      double xLowerLimitG1 = (1.0/slopeC2)*( positionG1.y - yInterceptC2 );
      double yLowerLimitG1 = slopeC1*positionG1.x + yInterceptC1;

      double xUpperLimitG1 = (1.0/slopeC2)*( positionG1.y - yIntercept2C12);
      double yUpperLimitG1 = slopeC1*positionG1.x + yIntercept1C12;

      double xLowerLimitG2 = (1.0/slopeC2)*( positionG2.y - yInterceptC2 );
      double yLowerLimitG2 = slopeC1*positionG2.x + yInterceptC1;

      double xUpperLimitG2 = (1.0/slopeC2)*( positionG2.y - yIntercept2C12);
      double yUpperLimitG2 = slopeC1*positionG2.x + yIntercept1C12;

      if ( positionG1.x >= xLowerLimitG1 && positionG1.x <= xUpperLimitG1 )
      {
    if ( positionG1.y >= yLowerLimitG1 && positionG1.y <= yUpperLimitG1 )
    {
      nAtomsPerLayer++;
    }
      }

      if ( positionG2.x >= xLowerLimitG2 && positionG2.x <= xUpperLimitG2 )
      {
    if ( positionG2.y >= yLowerLimitG2 && positionG2.y <= yUpperLimitG2 )
    {
      nAtomsPerLayer++;
    }
      }
    }
  }

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//    //account for missing atoms here, which must be checked by hand, unless we write a flag
here for atoms that are very close to the boundary
//  nAtomsPerLayer++;
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  return nAtomsPerLayer;
}



void assignGrapheneParams(AtomParams **Atom, struct vect a1, struct vect a2, struct vect a3,
struct vect b1, struct vect b2, struct vect b3)
{
  struct vect B1, B2, B12, C1, C2, C12;

    // Define TSB super-lattice vectors
  B1 = 4.0*b1;
  B2 = 4.0*b2;
  B12 = B1 + B2;
```

```
  // Define boundaries of TSB super-lattice vectors (purely for calculation)
C1 = B1 + 2.0*b3;
C2 = B2 + 2.0*b3;
C12 = B12 + 2.0*b3;

double slopeC1 = (C1.y - 2.0*b3.y) / (C1.x - 2.0*b3.x);
double slopeC2 = (C2.y - 2.0*b3.y) / (C2.x - 2.0*b3.x);
double yInterceptC1 = C1.y - slopeC1*C1.x;
double yInterceptC2 = C2.y - slopeC2*C2.x;

double yIntercept1C12 = C12.y - slopeC1*C12.x;
double yIntercept2C12 = C12.y - slopeC2*C12.x;

  // Initialize graphene basis vectors, where "1" and "2" denote basis atoms of the graphene
  unit cell
struct vect positionG1 = 2.0*b3;
struct vect positionG2 = 2.0*b3;

  // Initialize a sufficiently large grid to scan for atoms inside the boundaries
int maxUnitCells1 = 500;
int maxUnitCells2 = 500;

  //assign appropriate graphene atom positions and atom indices to class AtomParams
int atomCount = 0;
for (int i=-maxUnitCells1; i<maxUnitCells1; i++)
{
  for (int j=-maxUnitCells2; j<maxUnitCells2; j++)
  {
    positionG1 = i*a1 + j*a2 + a3;
    positionG2 = i*a1 + j*a2 + 2.0*a3;

    double xLowerLimitG1 = (1.0/slopeC2)*( positionG1.y - yInterceptC2 );
    double yLowerLimitG1 = slopeC1*positionG1.x + yInterceptC1;

    double xUpperLimitG1 = (1.0/slopeC2)*( positionG1.y - yIntercept2C12);
    double yUpperLimitG1 = slopeC1*positionG1.x + yIntercept1C12;

    double xLowerLimitG2 = (1.0/slopeC2)*( positionG2.y - yInterceptC2 );
    double yLowerLimitG2 = slopeC1*positionG2.x + yInterceptC1;

    double xUpperLimitG2 = (1.0/slopeC2)*( positionG2.y - yIntercept2C12);
    double yUpperLimitG2 = slopeC1*positionG2.x + yIntercept1C12;

    if ( positionG1.x >= xLowerLimitG1 && positionG1.x <= xUpperLimitG1 )
    {
  if ( positionG1.y >= yLowerLimitG1 && positionG1.y <= yUpperLimitG1 )
  {
    Atom[atomCount] = new AtomParams(positionG1, atomCount+1);
    atomCount++;
  }
    }
```

```
      if ( positionG2.x >= xLowerLimitG2 && positionG2.x <= xUpperLimitG2 )
      {
    if ( positionG2.y >= yLowerLimitG2 && positionG2.y <= yUpperLimitG2 )
    {
      Atom[atomCount] = new AtomParams(positionG2, atomCount+1);
      atomCount++;
    }
      }
    }
  }

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//    //account for missing atoms here
//  struct vect missingAtomPosition;
//  missingAtomPosition.x = -2.507;
//  missingAtomPosition.y = 46.151;
//  missingAtomPosition.z = 0.000;
//
//  Graphene[atomCount] = new AtomParams(missingAtomPosition, atomCount+1);
//  atomCount++;
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
}



void writePDB(AtomParams **Atom, int nCarbon, int nQuarter, int segmentID, double Bmag, double
domainAngle, int nAtomsPerLayer)
{
  stringstream outStream;
  outStream << "tsb35-c" << nCarbon << "_0" << nQuarter << "-quarter_LAY" << segmentID <<
  "-PDB.pdb";
  ofstream OUT(outStream.str().c_str());
  OUT << setiosflags(ios::fixed) << setprecision(3);

  struct vect position;
  string atomName;
  int atomIndex;

  OUT << "CRYST1   " << Bmag << "   " << Bmag << "  100.000  " << domainAngle << "   " <<
  domainAngle << "   " << domainAngle << "           P 1           1" << endl;

  for (int i=0; i<nAtomsPerLayer; i++)
  {
    stringstream ss;

    if (i < 1000) ss << "A" << i;
    else if (i < 2000) ss << "B" << (i-1000);
    else if (i < 3000) ss << "C" << (i-2000);
    else if (i < 4000) ss << "D" << (i-3000);
    else if (i < 5000) ss << "E" << (i-4000);
    else if (i < 6000) ss << "F" << (i-5000);
    else if (i < 7000) ss << "G" << (i-6000);
    else if (i < 8000) ss << "H" << (i-7000);
```

```
    else if (i < 9000) ss << "I" << (i-8000);
    else if (i < 10000) ss << "J" << (i-9000);
    else if (i < 11000) ss << "K" << (i-10000);
    else if (i < 12000) ss << "L" << (i-11000);
    else if (i < 13000) ss << "M" << (i-12000);
    else if (i < 14000) ss << "N" << (i-13000);
    else if (i < 15000) ss << "O" << (i-14000);
    else if (i < 16000) ss << "P" << (i-15000);
    else if (i < 17000) ss << "Q" << (i-16000);
    else if (i < 18000) ss << "R" << (i-17000);
    else if (i < 19000) ss << "S" << (i-18000);
    else if (i < 20000) ss << "T" << (i-19000);
    else if (i < 21000) ss << "U" << (i-20000);

    atomName=ss.str();

    position = Atom[i]->getPosition();
    Atom[i]->setPosition(position);

    OUT << "ATOM " << setw(6) << i+1;
    OUT << " " << setw(4) << atomName << " GRAP    1     ";
    OUT << setw(7) << position.x << " " << setw(7) << position.y << " " << setw(7) << position.z
     << "   ";
    OUT << "0.00  1.00      " << "LAY" << segmentID << endl;
  }

}



void Translate(AtomParams **Atom, struct vect translation, int nAtomsPerLayer)
{
  struct vect tempPosition, position;

  for (int i=0; i<nAtomsPerLayer; i++)
  {
    position = Atom[i]->getPosition();
    tempPosition = position + translation;
    Atom[i]->setPosition(tempPosition);
  }
}



int main(int argc, char *argv[])
{
    ////////////////
    // USER INPUT //
    ////////////////
  if (argc != 3) {
    cout << "Must have 2 command line arguments (nCarbon, nQuarter)!" << endl;
  }
```

```
int nCarbon = atoi(argv[1]);
int nQuarter = atoi(argv[2]);

float n1, n2;
double latticeSpacing, domainAngle;
if (nCarbon == 6) {

  n1 = 11.0 + float(nQuarter)*0.25;
  n2 = 3.0 + float(nQuarter)*0.25;

  if (nQuarter == 0) {
    latticeSpacing = 31.401;
    domainAngle = 11.742;
  } else if (nQuarter == 1) {
    latticeSpacing = 32.414;
    domainAngle = 12.331;
  } else if (nQuarter == 2) {
    latticeSpacing = 33.430;
    domainAngle = 12.885;
  } else if (nQuarter == 3) {
    latticeSpacing = 34.450;
    domainAngle = 13.407;
  } else if (nQuarter == 4) {
    latticeSpacing = 35.472;
    domainAngle = 13.898;
  } else {}

} else if (nCarbon == 10) {

  n1 = 15.0 + float(nQuarter)*0.25;
  n2 = 1.0 + float(nQuarter)*0.25;

  if (nQuarter == 0) {
    latticeSpacing = 38.182;
    domainAngle = 3.198;
  } else if (nQuarter == 1) {
    latticeSpacing = 39.135;
    domainAngle = 3.901;
  } else if (nQuarter == 2) {
    latticeSpacing = 40.095;
    domainAngle = 4.571;
  } else if (nQuarter == 3) {
    latticeSpacing = 41.059;
    domainAngle = 5.209;
  } else if (nQuarter == 4) {
    latticeSpacing = 42.028;
    domainAngle = 5.818;
  } else {}

} else {}

  ////////////////////////////
  // Build lattice vectors //
```

```
  ////////////////////////////
struct vect a1, a2, a3, b1, b2, b3;
double a = sqrt(3.0)*1.42;
double triAngle = 60.0*(PI/180.0);

  // Define graphene lattice vectors
a1.x = a;
a1.y = 0.0;
a1.z = 0.0;
a2.x = a*cos(triAngle);
a2.y = a*sin(triAngle);
a2.z = 0.0;
a3 = (1.0/3.0)*(a1+a2);

  // Define TSB lattice vectors
b1 = n1*a1 + n2*a2;
b2 = (-1.0)*n2*a1 + (n1+n2)*a2;
b3 = (-1.0/3.0)*(b1+b2);

double Bmag = 4.0*sqrt(b1.x*b1.x + b1.y*b1.y + b1.z*b1.z);

  // Use boundaries defined by TSB lattice vectors to calculate atom per graphene alyer
int nAtomsPerLayer = countGrapheneAtoms(a1, a2, a3, b1, b2, b3);
  // Create class for graphene atoms
AtomParams *Atom[nAtomsPerLayer];

cout << "Graphene parameters read and stored in class AtomParams/Atom." << endl;

  // Call to function to go back through grid and assign positions and atom names to the
  graphene atoms
assignGrapheneParams(Atom, a1, a2, a3, b1, b2, b3);

struct vect position, layerTranslate;
layerTranslate.x = 0.0;
layerTranslate.y = 0.0;
layerTranslate.z = -3.366;

int nALayers = 3;
int nBLayers = 3;
int segmentID;

  //write odd-numbered (A) layers
for (int i=0; i<nALayers; i++)
{
  Translate(Atom, 2*i*layerTranslate, nAtomsPerLayer);

  segmentID = 2*i+1;
  writePDB(Atom, nCarbon, nQuarter, segmentID, Bmag, domainAngle, nAtomsPerLayer);

  Translate(Atom, (-1)*2*i*layerTranslate, nAtomsPerLayer);
}

struct vect ABshift;
```

```
  ABshift.x = 1.23;
  ABshift.y = 0.0;
  ABshift.z = 0.0;

  Translate(Atom, ABshift, nAtomsPerLayer);

    //write even-numbered (B) layers, offset by 1.23 angstroms from odd-numbered layers
  for (int i=0; i<nBLayers; i++)
  {
    Translate(Atom, (2*i+1)*layerTranslate, nAtomsPerLayer);

    segmentID = 2*i+2;
    writePDB(Atom, nCarbon, nQuarter, segmentID, Bmag, domainAngle, nAtomsPerLayer);

    Translate(Atom, (-1)*(2*i+1)*layerTranslate, nAtomsPerLayer);
  }

  return 0;
}
```