#include<iostream>

#include<fstream>

#include<vector>

#include<iterator>

#include<math.h>

#include<string>

#include <sstream>

// this utility code reads 1D measured data, simulated data, and gives out the passing rate, the algorithm implemented is based on

// the method described in Depuydt et al.

// "A quantitative evaluation of IMRT dose distributions: refinement and clinical assessment of the gamma evaluation"

// Radiother Oncol 2002

// Important to note is that the measured data are the reference point,

// in other words we use each point in the measured dataset as the origin of coordinates

// and then we do the gamma analysis one-by-one for all the simulated points in the neighborhood of the reference measured point

// it is assumed that simulated data are high resolution data

// note that if you use the high resolution simulated data as the reference for the gamma analysis, many points will fail

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// This function reads the data from a file and converts it to a 2d vector //

//

// The data are tabulated in the file such that each line contains the distance first //

//

// ten a tab then the dose, no empty lines are allowed, empty line marks the end of the file //

// //

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void read(std::string filename, std::vector<std::vector<double>> & vec){

std::ifstream myFile;

myFile.open(filename);

if ( myFile.fail() ) {

// check if the file is in the folder, if not the code will terminate

std::cout<<"The entered filename could't be found!! Exiting the application.."<<std::endl;

exit(2);

}

std::string distance,dose;

while (myFile.good()){

std::vector<double> temp;

// read the first number in each line as the distance, the read is limited to the first tab

getline(myFile,distance,'\t');

// if the string being read is empty, the reading process will end

if (distance.empty()) break;

// convert the sring to double and store it in the vector container

temp.push\_back(stod(distance));

// read the second number in each line as the dose, the read is limited to the new line

getline(myFile,dose,'\n');

temp.push\_back(stod(dose));

vec.push\_back(temp);

}

}

//===================================================================================================================================

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// //

// A Normalizing function that takes a 2d vector (by reference) and a normalizing value as arguments //

// //

// and divide all the values in the 2nd column by the normalizing value //

//

//

// Note the use of -> begin()+1 //

//

// It is coded this way intentially for future ease of use with 2D and 3D gamma analysis //

// //

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void normalize\_vector(std::vector<std::vector<double>> &vector\_input, double &val) {

for (std::vector<std::vector<double>>::iterator norm\_itr = vector\_input.begin(); norm\_itr != vector\_input.end(); norm\_itr++)

\*(norm\_itr->begin()+1) = \*(norm\_itr->begin()+1) / val;

}

//===================================================================================================================================

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

///

// The interpolator function takes 2 arguments, an input depth, and a 2d vector of data in the form (distance, dose)

//

// It interpolates the depth between the values of the depth within the 1st column of 2d vector

//

//

// and gives the corresponding value of the dose

//

// The method used for searching here is the binary search algorithm assuming that the input dataset is larger

//

// so a lot of time will be saved instead of the sequential search

//

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

double interpolator (double &val, std::vector<std::vector<double>> &input\_vector){

std::vector<std::vector<double>>::iterator it1 = input\_vector.begin(); // 1st vector in the 2d vector

std::vector<std::vector<double>>::iterator it2 = prev(input\_vector.end()); // last vector in the 2d vector

std::vector<std::vector<double>>::iterator it\_mid; // mid iterator to be used when searching

double interpolated\_value; // the value to be returned

if (val < \*(it1->begin())) // if the depth is less than the 1st value given in data, we will extrapolate

interpolated\_value = \*(it1->begin()+1) - ( \*(next(it1,1)->begin()+1)-\*(it1->begin()+1) ) / ( \*(next(it1,1)->begin())-\*(it1->begin()) ) \* ( \*(it1->begin())-val );

else if (val > \*(it2->begin())) // if the depth is greater than the last value given in data, we will extrapolate

interpolated\_value = \*(it2->begin()+1) + ( \*(it2->begin()+1)-\*(prev(it2,1)->begin()+1) ) / ( \*(it2->begin())-\*(prev(it2,1)->begin()) ) \* ( val-\*(it2->begin()) );

else if (val == \*(it1->begin())) // if the depth is equal to the 1st value given in data, we will return the corresponding dose

interpolated\_value = \*(it1->begin()+1);

else if (val == \*(it2->begin())) // if the depth is equal to the last value given in data, we will return the corresponding dose

interpolated\_value = \*(it2->begin()+1);

else { // that must be a case in which the depth is within the given depths in the dataset

bool found\_in\_mid = false;

while (it2-it1 != 1){

it\_mid = it1 + (it2-it1)/2;

if (\*(it\_mid->begin()) == val) {

found\_in\_mid = true;

break; // break if the value of the input depth is found exactly in one of the depths given in the dataset

}

else if (\*(it\_mid->begin()) < val)

it1 = it\_mid;

else

it2 = it\_mid;

}

if (found\_in\_mid) // if the value of the input depth is exactly one of the dataset depths then read the corresponding dose

interpolated\_value = \*(it\_mid->begin()+1);

else //the depth is between 2 values and we will intepolate

interpolated\_value = \*(it1->begin()+1) + ( \*(it2->begin()+1)-(\*it1->begin()+1) ) / ( \*(it2->begin())-(\*it1->begin()) ) \* ( val-\*(it1->begin()) );

}

return interpolated\_value;

}

//=============================================================================================================================================================================

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

//

// The finder function

//

// GREATER OR EQUAL

//

// it has an input argument of depth (which is the reference measured point depth - DTA)

//

// the other input is the simulated dataset 2d vector

//

// the return value is an iterator to the simulated vector whose depth value is greater or equal to the input depth value.

//

// The reason for choosing greater or equal is that we need the simulated datapoints that are within the neighborhood

//

// of the measured point under consideration

//

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

std::vector<std::vector<double>>::iterator greater\_or\_equal\_finder (double &val, std::vector<std::vector<double>> &input\_vector){

std::vector<std::vector<double>>::iterator it1=input\_vector.begin();

std::vector<std::vector<double>>::iterator it2=prev(input\_vector.end());

std::vector<std::vector<double>>::iterator it\_mid;

if (\*(it1->begin()) >= val) // the 1st depth in the simulated data is greater than or equal to the input depth

return it1;

else if (\*(it2->begin()) == val) // the last depth in the simulated data is equal to the input depth

return it2;

else if (\*(it2->begin()) < val) { // the last depth in the simulated data is less than the input depth

std::cout << "There is no point in the simulated data that lies within the neighborhood of the reference measured point" << std::endl;

std::cout << "The reference measured point is outside the range of the simulated data and located at its right" << std::endl;

exit(2);

}

else { // that must be a case in which the input depth is within the depths of the simulated data

bool found\_in\_mid = false;

while (it2-it1 != 1){

it\_mid = it1 + (it2-it1)/2;

if (\*(it\_mid->begin()) == val) {

found\_in\_mid = true;

break;

}

else if (\*(it\_mid->begin()) < val)

it1 = it\_mid;

else

it2 = it\_mid;

}

if (found\_in\_mid)

return it\_mid;

else

return it2; // it2 becuase we need a value that is greater than the input depth

}

}

//==========================================================================================================================================================================

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

//

// The finder function

//

// LESS OR EQUAL

//

// it has an input argument of depth (which is the reference measured point depth + DTA)

//

// the other input is the simulated dataset 2d vector

//

// the return value is an iterator to the simulated vector whose depth value is less or equal to the input depth value.

//

// The reason for choosing les or equal is that we need the simulated datapoints that are within the neighborhood

//

// of the measured point under consideration

//

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

std::vector<std::vector<double>>::iterator less\_or\_equal\_finder (double &val, std::vector<std::vector<double>> &input\_vector){

std::vector<std::vector<double>>::iterator it1=input\_vector.begin();

std::vector<std::vector<double>>::iterator it2=prev(input\_vector.end());

std::vector<std::vector<double>>::iterator it\_mid;

if (\*(it1->begin()) > val){ // the first depth in the simulated data is greater than the input depth

std::cout << "There is no point in the simulated data that lies within the neighborhood of the reference measured point" << std::endl;

std::cout << "The reference measured point is outside the range of the simulated data and located at its left" << std::endl;

exit(2);

}

else if (\*(it1->begin()) == val) // the first depth in the simulated data is exactly equal to the input depth

return it1;

else if (\*(it2->begin()) <= val) // the last depth in the simulated data is less than or equal to the input depth

return it2;

else { // that must be a case in which the input depth is within the depths of the simulated data

bool found\_in\_mid = false;

while (it2-it1 != 1){

it\_mid = it1 + (it2-it1)/2;

if (\*(it\_mid->begin()) == val) {

found\_in\_mid = true;

break;

}

else if (\*(it\_mid->begin()) < val)

it1 = it\_mid;

else

it2 = it\_mid;

}

if (found\_in\_mid)

return it\_mid;

else

return it1; // it1 becuase we need a value that is less than the input depth

}

}

//==================================================================================================================================================

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//

// This function checks if the measured data passes the gamma analysis based on a given

//

// distance to agreement and dose criteria when compared with the simulated data

//

// The function also has arguments of lower and upper bounds of certqin ROI specified

//

// and a dose threshold (so that we are not checking very low dose values)

//

//

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

double gamma\_1D\_pass(std::vector<std::vector<double>> &measured\_vector, std::vector<std::vector<double>> &simulated\_vector,double &DTA, double &dose\_criteria,double &ROI\_beg, double &ROI\_end, double &dose\_threshold,int &pass, int &examined\_points){

pass = 0;

examined\_points = 0;

double temp\_min\_distance, temp\_max\_distance,temp\_max\_dose,temp\_min\_dose;

double a,b,c,m;

b = dose\_criteria\*0.01;

a = DTA;

bool check\_positive, check\_negative,pass\_check;

bool level2 = true;

bool level3 = true;

bool level4 = true;

int countx = 0;

//check measured data whether they pass or not

for (std::vector<std::vector<double>>::iterator measured\_itr = measured\_vector.begin(); measured\_itr != measured\_vector.end(); measured\_itr++){

countx += 1;

std::cout << countx;

if (\*(measured\_itr->begin()) > 10)

{

std::cout << std::endl<<\*(measured\_itr->begin())<<std::endl<< \*(measured\_itr->begin()+1)<<std::endl;

}

//cout<<simulated\_vector[i][1]<<endl;

if ((\*(measured\_itr->begin()) < ROI\_beg) || (\*(measured\_itr->begin()) > ROI\_end) || ( \*(measured\_itr->begin()+1) < dose\_threshold))

continue;

examined\_points++;

temp\_max\_distance=\*(measured\_itr->begin())+DTA;

temp\_min\_distance=\*(measured\_itr->begin())-DTA;

std::vector<std::vector<double>>::iterator start\_simulated\_itr = greater\_or\_equal\_finder (temp\_min\_distance, simulated\_vector);

std::vector<std::vector<double>>::iterator end\_simulated\_itr = less\_or\_equal\_finder (temp\_max\_distance, simulated\_vector);

pass\_check = false;

// In this case, only 1 simulated point is in the neighborhood, and this point is exactly the beginning of the simulation data

// This happens when the neighborhood is touching the first point of the simulation data or just lying a little within it

if (start\_simulated\_itr == end\_simulated\_itr && start\_simulated\_itr==simulated\_vector.begin() ){

if ( sqrt(pow( (\*(measured\_itr->begin())-\*(start\_simulated\_itr->begin()) ) / DTA,2) + pow(((\*(start\_simulated\_itr->begin()+1) / \*(measured\_itr->begin()+1))-1) / (0.01\*dose\_criteria),2)) <= 1.0 ) {

pass++;

pass\_check=true;

}

//check semi level 3

if ( (!pass\_check) && (level3) ){

m=(\*(next(start\_simulated\_itr,1)->begin()+1) - \*(start\_simulated\_itr->begin()+1))/(\*(next(start\_simulated\_itr,1)->begin()) - \*(start\_simulated\_itr->begin()));

c= \*(start\_simulated\_itr->begin()+1) - \*(measured\_itr->begin()+1) - m \* ( \*(start\_simulated\_itr->begin()) - \*(measured\_itr->begin()) );

if ( b\*b+a\*a\*m\*m-c\*c >=0 ){

pass++;//level 3 has passed

pass\_check=true;

}//pass level 3

}//end of semi level 3

}// end of if

//only 1 point in the neighborhood, and the neighborhood is at the end of the simulation data

//may be only the last point of the simulation data is just at the extremity of neighborhood OR a little within it

else if (start\_simulated\_itr == end\_simulated\_itr && start\_simulated\_itr == prev(simulated\_vector.end())){

if ( sqrt(pow( (\*(measured\_itr->begin())-\*(end\_simulated\_itr->begin()) ) / DTA,2) + pow(((\*(end\_simulated\_itr->begin()+1) / \*(measured\_itr->begin()+1))-1) / (0.01\*dose\_criteria),2)) <= 1.0 ) {

pass++;

pass\_check=true;

}

//check semi level 3

if ( (!pass\_check) && (level3) ){

m=(\*(prev(end\_simulated\_itr,1)->begin()+1) - \*(end\_simulated\_itr->begin()+1))/(\*(prev(end\_simulated\_itr,1)->begin()) - \*(end\_simulated\_itr->begin()));

c= \*(end\_simulated\_itr->begin()+1) - \*(measured\_itr->begin()+1) - m \* ( \*(end\_simulated\_itr->begin()) - \*(measured\_itr->begin()) );

if ( b\*b+a\*a\*m\*m-c\*c >=0 ){

pass++;//level 3 has passed

pass\_check=true;

}//pass level 3

}//end of semi level 3

}//end of else if

//only 1 point in the neighborhood, and the neighborhood is within the simulation data

else if (start\_simulated\_itr == end\_simulated\_itr&& start\_simulated\_itr != simulated\_vector.begin() && start\_simulated\_itr != prev(simulated\_vector.end())){

if ( sqrt(pow( (\*(measured\_itr->begin())-\*(end\_simulated\_itr->begin()) ) / DTA,2) + pow(((\*(end\_simulated\_itr->begin()+1) / \*(measured\_itr->begin()+1))-1) / (0.01\*dose\_criteria),2)) <= 1.0 ) {

pass++;

pass\_check=true;

}

//check semi level 3

if ( (!pass\_check) && ( level3 ) ){

m=(\*(next(start\_simulated\_itr,1)->begin()+1) - \*(start\_simulated\_itr->begin()+1))/(\*(next(start\_simulated\_itr,1)->begin()) - \*(start\_simulated\_itr->begin()));

c= \*(start\_simulated\_itr->begin()+1) - \*(measured\_itr->begin()+1) - m \* ( \*(start\_simulated\_itr->begin()) - \*(measured\_itr->begin()) );

if ( b\*b+a\*a\*m\*m-c\*c >=0 ){

pass++;//level 3 has passed

pass\_check=true;

}//pass level 3

}//end of semi level 3

//check semi level 3

if ( (!pass\_check) && ( level3 ) ){//2nd if of the left side

m=(\*(prev(end\_simulated\_itr,1)->begin()+1) - \*(end\_simulated\_itr->begin()+1))/(\*(prev(end\_simulated\_itr,1)->begin()) - \*(end\_simulated\_itr->begin()));

c= \*(end\_simulated\_itr->begin()+1) - \*(measured\_itr->begin()+1) - m \* ( \*(end\_simulated\_itr->begin()) - \*(measured\_itr->begin()) );

if ( b\*b+a\*a\*m\*m-c\*c >=0 ){

pass++;//level 3 has passed

pass\_check=true;

}//pass level 3

}//end of semi level 3

}//end of else if

//the neighborhood is between 2 simulated points and does not have any points within it//// \_\_\_\_ ONLY LEVEL 4 \_\_\_\_ ///////

else if (start\_simulated\_itr > end\_simulated\_itr){

//check level 4

if ( level4 ){

m=(\*(start\_simulated\_itr->begin()+1) - \*(end\_simulated\_itr->begin()+1))/(\*(start\_simulated\_itr->begin()) - \*(end\_simulated\_itr->begin()));

c= \*(end\_simulated\_itr->begin()+1) - \*(measured\_itr->begin()+1) - m \* ( \*(end\_simulated\_itr->begin()) - \*(measured\_itr->begin()) );

if ( b\*b+a\*a\*m\*m-c\*c >=0 ){

pass++;//level 3 has passed

pass\_check=true;

}//pass level 3

}//end of level 4

}//end of else if

//normal neighborhood containing different points

else {

//normal regions

//start to check level 1

for (std::vector<std::vector<double>>::iterator simulated\_itr = start\_simulated\_itr; simulated\_itr <= end\_simulated\_itr; simulated\_itr++) {

if ( sqrt(pow( (\*(measured\_itr->begin())-\*(simulated\_itr->begin()) ) / DTA,2) + pow(((\*(simulated\_itr->begin()+1) / \*(measured\_itr->begin()+1))-1) / (0.01\*dose\_criteria),2)) <= 1.0 ) {

pass++;

pass\_check=true;

break;

}

}// end of for

//start to check level 2

check\_positive= false;

check\_negative= false;

if ( ! pass\_check && level2 ){

for (std::vector<std::vector<double>>::iterator simulated\_itr = start\_simulated\_itr; simulated\_itr <= end\_simulated\_itr; simulated\_itr++) {

if (!check\_positive){

if( \*(simulated\_itr->begin()+1) > \*(measured\_itr->begin()+1) ) check\_positive=true;

}// ne equality, since if it was equal to zero it should have been passed :)

if (!check\_negative){

if ( \*(simulated\_itr->begin()+1) < \*(measured\_itr->begin()+1) ) check\_negative=true;

}

if ((check\_negative) && (check\_positive)){

pass++;//level 2 has passed

pass\_check=true;

break;

}

}// end of for

}//end of level 2

//check for level 3

if ( (! pass\_check) && (level3) && (end\_simulated\_itr != prev(simulated\_vector.end())) ){

//checker\_level3 ( measured\_itr, start\_simulated\_itr,end\_simulated\_itr, pass,dose\_criteria,DTA,pass\_check);

m=(\*(next(end\_simulated\_itr,1)->begin()+1) - \*(end\_simulated\_itr->begin()+1))/(\*(next(end\_simulated\_itr,1)->begin()) - \*(end\_simulated\_itr->begin()));

c= \*(end\_simulated\_itr->begin()+1) - \*(measured\_itr->begin()+1) - m \* ( \*(end\_simulated\_itr->begin()) - \*(measured\_itr->begin()) );

if ( b\*b+a\*a\*m\*m-c\*c >=0 ){

pass++;//level 3 has passed

pass\_check=true;

}//pass level 3

//in the second one we will add the pass check in case it passed the 1st point

if ( (! pass\_check) && (start\_simulated\_itr != simulated\_vector.begin()) )

{

float x2 = \*(start\_simulated\_itr->begin() + 1);

float y2 = \*(start\_simulated\_itr->begin());

float x3 = \*(prev(start\_simulated\_itr, 1)->begin());

float x4 = \*(start\_simulated\_itr->begin());

float x1 = \*(prev(start\_simulated\_itr, 1)->begin() + 1);

m=(\*(prev(start\_simulated\_itr,1)->begin()+1) - \*(start\_simulated\_itr->begin()+1))/(\*(prev(start\_simulated\_itr,1)->begin()) - \*(start\_simulated\_itr->begin()));

c= \*(start\_simulated\_itr->begin()+1) - \*(measured\_itr->begin()+1) - m \* ( \*(start\_simulated\_itr->begin()) - \*(measured\_itr->begin()) );

if ( b\*b+a\*a\*m\*m-c\*c >=0 ){

pass++;//level 3 has passed

pass\_check=true;

}//pass level 3

}//end 2nd if

}//end of level 3

}//end of else

}//end of check of measured data

return 100\*static\_cast<double>(pass)/examined\_points;

}

///==========================================================================================================================

////\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ POINT BY POINT \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//calculate dose difference point by point, do we need binary search for more efficient computation?

// here the difference is the simulated value - measured value

void point\_by\_point\_comparison(std::vector<std::vector<double>> &measured\_vector, std::vector<std::vector<double>> &simulated\_vector, double &ROI\_beg, double &ROI\_end, double &dose\_threshold){

std::vector<double> difference;

for (std::vector<std::vector<double>>::iterator measured\_itr = measured\_vector.begin(); measured\_itr != measured\_vector.end(); measured\_itr++){

if((\*(measured\_itr->begin()+1)< dose\_threshold ) || (\*(measured\_itr->begin()) < ROI\_beg) || (\*(measured\_itr->begin()) > ROI\_end))

{

difference.push\_back(0.);//to keep track of order

continue;

}

for(std::vector<std::vector<double>>::iterator simulated\_itr = simulated\_vector.begin(); simulated\_itr != simulated\_vector.end(); simulated\_itr++){

if (\*(simulated\_itr->begin()) == \*(measured\_itr->begin())){

difference.push\_back(\*(simulated\_itr->begin()+1)-\*(measured\_itr->begin()+1));

break;

}

else if ( (\*(simulated\_itr->begin()) < \*(measured\_itr->begin())) && (\*(next(simulated\_itr,1)->begin()) > \*(measured\_itr->begin())) ){

difference.push\_back(\*(simulated\_itr->begin()+1)+(\*(measured\_itr->begin())-\*(simulated\_itr->begin()))/(\*(next(simulated\_itr,1)->begin())-\*(simulated\_itr->begin()))\*(\*(next(simulated\_itr,1)->begin()+1)-\*(simulated\_itr->begin()+1))-\*(measured\_itr->begin()+1));

break;

}

}

}

std::ofstream outFile;

outFile.open("point\_by\_point\_results.txt");

for(int k=0; k<difference.size();k++) outFile<<" "<<difference.at(k);

outFile.close();

}//end of function

////\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_END OF POINT BY POINT\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

int main(){

std::string measured\_data\_file,simulated\_data\_file,string\_temp;

//double dose\_criteria=3.0;

//double DTA=0.3;

int modWheel=60;

double energy=149;

double dose\_threshold=.10;

//bool level2=false;

//bool level3=false;

//bool level4=false;

int pass=0;

int examined\_points=0;

/\*std::cout<<"Enter the name and extension of the measured data file "<<std::endl;

getline(std::cin,measured\_data\_file);

std::cout<<"Enter the name and extension of the simulated data file "<<std::endl;

getline(std::cin,simulated\_data\_file);

\*/

/\*

cout<<"Enter the dose criteria in % [DEFAULT is 3%]"<<endl;

getline(cin,string\_temp);

if(!string\_temp.empty())

dose\_criteria=stod(string\_temp);

cout<<"Enter the distance to agreement in cm [DEFAULT is 0.3 cm]"<<endl;

getline(cin,string\_temp);

if(!string\_temp.empty())

DTA=stod(string\_temp);

\*/

/\*std::cout<<"Enter the mod Wheel [DEFAULT is 60]"<<std::endl;

getline(std::cin,string\_temp);

if(!string\_temp.empty())

modWheel=stoi(string\_temp);

std::cout<<"Enter the energy [DEFAULT is 149]"<<std::endl;

getline(std::cin,string\_temp);

if(!string\_temp.empty())

energy=stod(string\_temp);\*/

std::vector<std::vector<double>> measured\_vector;

std::vector<std::vector<double>> simulated\_vector;

//read(measured\_data\_file, measured\_vector);

//read(simulated\_data\_file, simulated\_vector);

//read("126\_topas\_pdd2.csv", measured\_vector);

//read("126\_odyssey\_pdd2.csv", simulated\_vector);

read("126\_topas\_pdd3.csv", measured\_vector);

read("126\_odyssey\_pdd3.csv", simulated\_vector);

//to specify an ROI

int n = measured\_vector.size();

double ROI\_beg = measured\_vector.at(0).at(0);

double ROI\_end = measured\_vector.at(n-1).at(0);

std::cout<<"Enter the lower depth of ROI for which you want to do the calculation [DEFAULT is the 1st point]"<<std::endl;

getline(std::cin,string\_temp);

if(!string\_temp.empty())

ROI\_beg=stod(string\_temp);

std::cout <<"lower depth = "<< ROI\_beg << std::endl;

std::cout << "Enter the larger depth of ROI for which you want to do the calculation [DEFAULT is the end point]" << std::endl;

getline(std::cin, string\_temp);

if (!string\_temp.empty())

ROI\_end = stod(string\_temp);

std::cout <<"higher depth = "<< ROI\_end << std::endl;

double normalization\_depth\_measured\_data;

double normalization\_dose\_measured\_data;

//normalize the measured data or not

std::cout<<"Do you want to normalize the measured data (Enter y or n) [DEFAULT is NO]"<<std::endl;

getline(std::cin,string\_temp);

if(string\_temp == "y"){

std::cout<<"What is the depth at which you want to normalize the data in cm [THERE IS NO DEFAULT here!!!]"<<std::endl;

getline(std::cin,string\_temp);

if (string\_temp.empty()){

exit(2);

}

else {

normalization\_depth\_measured\_data=stod(string\_temp);

normalization\_dose\_measured\_data=interpolator (normalization\_depth\_measured\_data, measured\_vector);

normalize\_vector(measured\_vector,normalization\_dose\_measured\_data);

}

}

double normalization\_depth\_simulated\_data;

double normalization\_dose\_simulated\_data;

//normalize the simulated data or not

std::cout<<"Do you want to normalize the simulated data (Enter y or n) [DEFAULT is NO]"<<std::endl;

getline(std::cin,string\_temp);

if(string\_temp == "y"){

std::cout<<"What is the depth at which you want to normalize the data in cm [THERE IS NO DEFAULT here!!!]"<<std::endl;

getline(std::cin,string\_temp);

if (string\_temp.empty()){

exit(2);

}

else {

normalization\_depth\_simulated\_data=stod(string\_temp);

normalization\_dose\_simulated\_data=interpolator (normalization\_depth\_simulated\_data, simulated\_vector);

normalize\_vector(simulated\_vector,normalization\_dose\_simulated\_data);

}

}

//specify the dose threshold

std::cout<<"Enter the dose threshold below which there is no calculation [DEFAULT is 0.10], be careful if normalized!!"<<std::endl;

getline(std::cin,string\_temp);

if (!string\_temp.empty())

dose\_threshold=stod(string\_temp);

std::cout << "dose threshold = "<<dose\_threshold<<std::endl;

//Pass rate criteria in %

double pass\_rate=90.0;

std::cout<<"Enter the pass rate percentage in % {DEFAULT is 90%]"<<std::endl;

getline(std::cin,string\_temp);

if (! string\_temp.empty())

pass\_rate=stod(string\_temp);

std::cout << "pass rate = " << pass\_rate << std::endl;

bool point\_by\_point\_analysis=false;

std::cout<<"Do you want to perform point-by-point check? [DEFAULT is NO]"<<std::endl;

getline(std::cin, string\_temp);

if(!string\_temp.empty())

point\_by\_point\_analysis=true;

bool not\_passed = true;

int mm\_check=1;

double mm\_check\_float;

double dose\_criteria\_float;

double result\_percentage;

std::ofstream outFile2;

remove("Gamma\_detail.txt");

outFile2.open("Gamma\_detail.txt");

outFile2 << "energy " << energy << std::endl << "Mod Wheel " << modWheel << std::endl

<< "ROI" << " Bigin = " << ROI\_beg << " End = " << ROI\_end << std::endl;

std::cout << "energy " << energy << std::endl << "Mod Wheel " << modWheel << std::endl

<< "ROI" << " Bigin = " << ROI\_beg << " End = " << ROI\_end << std::endl;

//while (not\_passed) {

while (mm\_check < 4) {

mm\_check\_float=mm\_check; //mm\_check\_float=mm\_check\*0.1;

for(int i=1;i<4;i++){

dose\_criteria\_float=static\_cast<double>(i);

result\_percentage = gamma\_1D\_pass(measured\_vector, simulated\_vector,mm\_check\_float, dose\_criteria\_float,ROI\_beg, ROI\_end, dose\_threshold,pass,examined\_points);

std::cout << mm\_check\_float<< "mm and " << dose\_criteria\_float<< "% pass rate " <<result\_percentage << "%" << std::endl;

outFile2 << "The number of points examined is " << examined\_points << std::endl

<< "The number of points which passed the test is " << pass << std::endl

<< "And the passing percentage is " << result\_percentage << " %" << " and the least passing criteria is " << std::endl

<< mm\_check\_float << " mm and " << dose\_criteria\_float << "%" << std::endl;

if ( result\_percentage > pass\_rate )

{

not\_passed = false;

// break;

}

outFile2 << std::endl;

}

if (mm\_check<3)

{

mm\_check++;

}

else

{

break;

}

}

outFile2.close();

if (not\_passed) {

std::cout<<" The test did not pass even for 3 mm 3% "<<std::endl;

}

else{

std::ofstream outFile;

remove("Gamma\_results.txt");

outFile.open("Gamma\_results.txt");

outFile<<"energy "<< energy <<std::endl <<"Mod Wheel "<< modWheel <<std::endl <<"The number of points examined is "<< examined\_points <<std::endl<<"The number of points which passed the test is "<<pass<<std::endl<<"And the passing percentage is "<<result\_percentage<<" %"<<" and the least passing criteria is "<<mm\_check\_float<<" mm and "<<dose\_criteria\_float<<"%"<<std::endl;

outFile.close();

if (point\_by\_point\_analysis) point\_by\_point\_comparison(measured\_vector, simulated\_vector, ROI\_beg, ROI\_end, dose\_threshold);

}

}//End of MAIN