```
#include "twoPointBVP.h"


// A C++ class implementation for two point BVPs

TwoPointBVP::TwoPointBVP(double *dom, double(*dFunc) (vector<double> &))
{
        domain = dom;
        diffusion = dFunc;
        reactionIsPresent = false;
        forcingFunctIsPresent = false;
}

void TwoPointBVP::set_left_bdry(bool _leftIsDirichlet, double *val)
{
        leftBdryIsDirichlet = _leftIsDirichlet;
        leftBdryValues = val;
}

void TwoPointBVP::set_right_bdry(bool _rightIsDirichlet, double *val)
{
        rightBdryIsDirichlet = _rightIsDirichlet;
        rightBdryValues = val;
}

void TwoPointBVP::set_reaction(double(*rFunctOne) (vector<double> &),
                                                        double(*rFunctTwo) (v
ector<double> &))
{
        reaction = rFunctOne;
        partialreactionpartialu = rFunctTwo;
        reactionIsPresent = true;
}

void TwoPointBVP::set_forcing_function(double(*fFunct) (vector<double> &))
{
        forcingFunct = fFunct;
        forcingFunctIsPresent = true;
}

void TwoPointBVP::set_true_solution(double(*TrueSol)(vector<double>&))
{
        trueSolu = TrueSol;
        trueSolIsPresent = true;
}

double * TwoPointBVP::get_domain() const
{
        return domain;
}

bool TwoPointBVP::left_bdry_is_Dirichlet() const
{
        return leftBdryIsDirichlet;
}

bool TwoPointBVP::right_bdry_is_Dirichlet() const
{
        return rightBdryIsDirichlet;
```

```
}

double * TwoPointBVP::get_left_bdry_values() const
{
        return leftBdryValues;
}

double * TwoPointBVP::get_right_bdry_values() const
{
        return rightBdryValues;
}

bool TwoPointBVP::reaction_is_present() const
{
        return reactionIsPresent;
}

bool TwoPointBVP::forcing_fucntion_is_present() const
{
        return forcingFunctIsPresent;
}

bool TwoPointBVP::true_solution_is_present() const
{
        return trueSolIsPresent;
}

double TwoPointBVP::eval_diffusion(vector<double> &x) const
{
        return diffusion(x);
}

vector<double> TwoPointBVP::eval_reaction(vector<double> &par) const
{
        vector<double> val(2);
        val[0] = reaction(par);
        val[1] = partialreactionpartialu(par);
        return val;
}

double TwoPointBVP::eval_forcing_function(vector<double>& x) const
{
        return forcingFunct(x);
}

double TwoPointBVP::eval_true_solution(vector<double>& x) const
{
        return trueSolu(x);
}

void TwoPointBVP::display_info_TwoPointBVP() const
{
        ofstream fileout;
        fileout.open("problem_info.txt");
        fileout << "**********************************************************************
\n";
        fileout << " Some info regarding the two point BVP problem and approximation: \n";
```

```
        fileout << " Domain is:(" << domain[0] << "," << domain[1] << ")" << endl;

        if (leftBdryIsDirichlet)
        {
                fileout << " Left boundary is Dirichlet with value: " << leftBdryValues[1] <<
 endl;
        }
        else if (leftBdryValues[0] == 0)
        {
                fileout << " Left Boundary is Neumann with g_0: " << leftBdryValues[1] <<
endl;
        }
        else
        {
                fileout << " Left Boundary is Robin with gamma_0: " << leftBdryValues[0]
                        << " and g_0: " << leftBdryValues[1] << endl;
        }



        if (rightBdryIsDirichlet)
        {
                fileout << " Right boundary is Dirichlet with value: " << rightBdryValues[1] <
< endl;
        }
        else if (rightBdryValues[0] == 0)
        {
                fileout << " Right Boundary is Neumann with g_L: " << rightBdryValues[1] <
< endl;
        }
        else
        {
                fileout << " Right Boundary is Robin with gamma_L: " << rightBdryValues[0]
                        << " and g_L: " << rightBdryValues[1] << endl;
        }

        if (reactionIsPresent)
        {
                fileout << " Reaction is Present. \n";
        }
        else
        {
                fileout << " No reaction is present. \n";
        }

        if (forcingFunctIsPresent)
        {
                fileout << " Forcing Function is Present. \n";
        }
        else
        {
                fileout << " Forcing function is not present. \n";
        }

        fileout << "*****************************************************************
\n";
        fileout.close();
        return;
}
```

```
vector<double> TwoPointBVP::true_solution(int numEvals, double domLeft, double d
omRight)
{
        vector<double> trueSoln(numEvals);
        double steps = (domRight-domLeft) / (numEvals-1);
        for (int i = 0; i < numEvals; i++)
        {
                vector<double> value(1);
                value[0] = domLeft + i * steps;
                trueSoln[i] = eval_true_solution(value);
        }
        return trueSoln;
}

TwoPointBVP::~TwoPointBVP()
{
        ;
}
```