```
#include "tridiagonal_matrix.h"
#include "twoPointBVP.h"
#include "TwoPointBVPAppr.h"


TwoPointBVPAppr::TwoPointBVPAppr(int N, const double * subintervallengths, const
 TwoPointBVP * prob)
{
        numsubintervals = N;
        steplenghts = subintervallengths;
        theproblem = prob;

        // if either reaction or external force is present, then we need Delta x
_i
        if (theproblem→reaction_is_present() ∨ theproblem→forcing_fucntion_is_
present())
        {
                Deltax.resize(numsubintervals + 1);
                Deltax[0] = 0.5 * steplenghts[0];
                for (int i = 1; i < numsubintervals; i++)
                {
                        Deltax[i] = 0.5*steplenghts[i - 1] + 0.5*steplenghts[i];
                }

                Deltax[numsubintervals] = 0.5 * steplenghts[numsubintervals - 1]
;
        }
        //This is x_i
        xcoord.resize(numsubintervals + 1);
        double * domain = theproblem→get_domain();
        xcoord[0] = domain[0];
        for (int i = 1; i ≤ numsubintervals; i++)
        {
                xcoord[i] = xcoord[i - 1] + steplenghts[i - 1];
        }

        //This is x_{i -1/2} or x_{i + 1/2}
        midcoord.resize(numsubintervals + 2);
        midcoord[0] = domain[0];
        for (int i = 1; i ≤ numsubintervals; i++)
        {
                midcoord[i] = 0.5 * (xcoord[i] + xcoord[i - 1]);
        }
        midcoord[N + 1] = xcoord[numsubintervals];
}

vector<double> TwoPointBVPAppr::get_xcoord()
{
        return xcoord;
}

int TwoPointBVPAppr::get_numsubintervals()
{
        return  numsubintervals;
}

void TwoPointBVPAppr::set_intial_guess_seed(double(*guessSeed)(vector<double> &)
)
{
```

```
        intialGuessSeed = guessSeed;
        guess_seed_is_present = true;
}

double TwoPointBVPAppr::eval_intial_guess_seed(vector<double>& par) const
{
        return intialGuessSeed(par);
}

void TwoPointBVPAppr::AssembleDiffusion(tridiagonal_matrix *tmat)
{
        vector <double> kappa(numsubintervals);
        vector <double> par(1);
        for (int i = 0; i < numsubintervals; i++)
        {
                par[0] = midcoord[i+1];
                kappa[i] = theproblem→eval_diffusion(par);
        }


        for (int i = 0; i < numsubintervals; i++)
        {
                kappa[i] = kappa[i] / steplenghts[i];
        }

        // taking care of the zeroth row(left boundary)
        if (theproblem→left_bdry_is_Dirichlet())
        {
                // if the left boundary is Dirichlet then the
                //first row of the tridiagMat has coeffceint
                //1 for diag[1] and 0 for upperdiag[0]
                tmat→set_diagonal_entry(0, 1.0);
                tmat→set_upper_diagonal_entry(0, 0);
        }
        else
        {
                // if the left bdry is not Dirichelt it is Nueman or Robin
                //However if Nueman LBVal[0] = 0 (gamma = 0)
                double *LBV;
                LBV = theproblem→get_left_bdry_values();
                tmat→set_diagonal_entry(0, kappa[0] - LBV[0]);
                tmat→set_upper_diagonal_entry(0, -kappa[0]);
        }

        //filling up the matrix tmat for internal rows
        for (int i = 1; i ≤ numsubintervals -1; i++)
        {
                // the interior points are the coeffs in the
                // approximate soln given by eqaution star
                tmat→set_lower_diagonal_entry(i - 1, -kappa[i-1]);
                tmat→set_diagonal_entry(i, kappa[i] + kappa[i-1]);
                tmat→set_upper_diagonal_entry(i, -kappa[i]);
        }
        // taking care of the last row(right boundary)
        if (theproblem→right_bdry_is_Dirichlet())
        {
                // if the right bdry is Dirichlet then the
                // last row of the tridiagMat has coeff
                // 1 for diag[N] and 0 for lowerdiag[N-1]
```

```
                tmat->set_diagonal_entry(numsubintervals, 1.0);
                tmat->set_lower_diagonal_entry(numsubintervals - 1, 0);
        }
        else
        {
                // it is either nuemen or robin (if nueman then gamma_n = 0)
                double *RBV;
                RBV = theproblem->get_right_bdry_values();
                tmat->set_diagonal_entry(numsubintervals, kappa[numsubintervals-
1] - RBV[0]);
                tmat->set_lower_diagonal_entry(numsubintervals - 1, -kappa[numsu
bintervals-1]);
        }
}

void TwoPointBVPAppr::AssembleReaction(vector<double> &U,
                vector<double> &RW, vector<double> &RPW )
{

        //U is our Solution, RW is the new reaction force and RPW is the partial
 derivative of RW

        vector <double> par(2);
        vector <double> val(2);

        //taking care of the left boundary
        if (theproblem->left_bdry_is_Dirichlet())
        {
                //if dirichlet Reaction[0] = 0
                RW[0] = 0;
                RPW[0] = 0;
        }
        else
        {
                //if not dirichlet Reaction[0] will be evaluated for par[0] and
par2
                par[0] = xcoord[0];
                par[1] = U[0];
                val =  theproblem->eval_reaction(par);
                RW[0] = val[0] * Deltax[0];
                RPW[0] = val[1] * Deltax[0];

        }

        // now the middle points
        for (int i = 1; i < numsubintervals; i++)
        {
                par[0] = xcoord[i];
                par[1] = U[i];
                val = theproblem->eval_reaction(par);
                RW[i] = val[0] * Deltax[i];
                RPW[i] = val[1] * Deltax[i];
        }

        // now the right boundary
        if (theproblem->right_bdry_is_Dirichlet())
        {
                //fill this
                RW[numsubintervals] = 0;
```

```
                RPW[numsubintervals] = 0;
        }
        else
        {
                //filling the necessary conditions for the right boundary
                par[0]= xcoord[numsubintervals];
                par[1] = U[numsubintervals];
                val = theproblem→eval_reaction(par);
                RW[numsubintervals] = val[0] * Deltax[numsubintervals];
                RPW[numsubintervals] = val[1] * Deltax[numsubintervals];
        }

}

vector<double> TwoPointBVPAppr::AssembleForce()
{

        // This vector contains the force algebraic terms
        vector<double> FF(numsubintervals + 1);

        vector <double> par(1);

        //if there is a forcing function
        // set all terms equal to forcing function
        // or boundary conditions or both
        if (theproblem →forcing_fucntion_is_present())
        {
                // Left boundary
                if (theproblem→left_bdry_is_Dirichlet())
                {
                        // when it is dirichlet the rhs[0] is simply g_0
                        double *LBC = theproblem→get_left_bdry_values();
                        FF[0] = LBC[1];
                }
                else
                {
                        //if it is Nueman or robin then the rhs[0] is
                        // ff-g_0
                        double *LBC = theproblem→get_left_bdry_values();
                        par[0] = xcoord[0];
                        FF[0] = theproblem→eval_forcing_function(par)*Deltax[0]
 - LBC[1];
                }

                //interior points
                for (int i = 1; i < numsubintervals; i++)
                {
                        //for the interior points the FF is governed soley by it
self
                        par[0] = xcoord[i];
                        FF[i] = theproblem→eval_forcing_function(par)*Deltax[i]
;

                }


                // Right boundary
                if (theproblem→right_bdry_is_Dirichlet())
                {
```

```
                        //if it is Dirichlet then the rhs is g_L
                        double *RBC = theproblem→get_right_bdry_values();
                        FF[numsubintervals] = RBC[1];
                }
                else
                {
                        //if it is Nueman or Robin then the rhs is FF-g_L
                        double *RBC = theproblem→get_right_bdry_values();
                        par[0] = xcoord[numsubintervals];
                        FF[numsubintervals] = theproblem→eval_forcing_function(
par)*Deltax[numsubintervals] - RBC[1];
                }
        }

        //if there isnt a ForcingFunct then set FF all
        //equal to zero except when otherwise dictated by BCs.
        else
        {
                        // Left boundary
                if (theproblem→left_bdry_is_Dirichlet())
                {
                        // when it is dirichlet the rhs[0] is simply g_0
                        double *LBC = theproblem→get_left_bdry_values();
                        FF[0] = LBC[1];
                }
                else
                {
                        //if it is Nueman or robin then the rhs[0] is
                        // ff-g_0
                        double *LBC = theproblem→get_left_bdry_values();
                        FF[0] =  -LBC[1];
                }

                //interior points
                for (int i = 1; i < numsubintervals; i++)
                {
                        //for the interior FF = 0
                        FF[i] = 0;
                }

                // Right boundary
                if (theproblem→right_bdry_is_Dirichlet())
                {
                        //if it is Dirichlet then the rhs[N] is g_L
                        double *RBC = theproblem→get_right_bdry_values();
                        FF[numsubintervals] = RBC[1];
                }
                else
                {
                        //if it is Nueman or Robin then the rhs is -g_L
                        double *RBC = theproblem→get_right_bdry_values();
                        FF[numsubintervals] = - RBC[1];
                }
        }

        return FF;
}

double find_l2_norm(vector<double> const x)
```

```
{
        int num_entries = x.size();
        double sum_o_squares = 0;
        for (int i = 0; i < num_entries ; i++)
        {
                sum_o_squares = sum_o_squares + x[i] * x[i];
        }
         return sqrt(sum_o_squares);
}


vector<double> TwoPointBVPAppr::Solve(int max_num_iter, double TOL)
{
        int iteration_counter = 0;
        double norm;
        tridiagonal_matrix *Gp, *A;
        vector<double> R(numsubintervals + 1, 0.0);
        vector<double>Rp(numsubintervals + 1, 0.0);
        vector<double>F;
        A = new tridiagonal_matrix(numsubintervals + 1);
        // Calculate the tridiagonal matrix coming from diffusion component.
        AssembleDiffusion(A);
        // Create the forcing function
        F = AssembleForce();

        //Create intial guess of Soln vector U
        vector<double> U(numsubintervals + 1, 3.0);
        //if there is a seed function present use it to form
        //the intial guess for the U solution vector.
        if (guess_seed_is_present)
        {
                vector<double> par(1);
                vector<double> U(numsubintervals + 1, 3.0);
                U[0] = F[0];
                for (int i = 1; i < numsubintervals; i++)
                {
                        par[0] = xcoord[i];
                        U[i] = eval_intial_guess_seed(par);
                }
                U[numsubintervals] = F[numsubintervals];
        }
        //if a seed isn't present set all interior points to the same number.
        else
        {
                U[0] = F[0];
                U[numsubintervals] = F[numsubintervals];
        }



        vector<double> h(numsubintervals + 1, 0.0);
        vector<double> G(numsubintervals + 1, 0.0);
        vector<double> AU(numsubintervals + 1);



        // The iteration
        for (int iter = 1; iter ≤ max_num_iter; iter++)
        {
```

```
                // Copy A to Gp
                Gp = new tridiagonal_matrix(A);

                // if there is a reaction calculate r(x,u) and pd(r(x,u),u)
                if (theproblem→reaction_is_present())
                {
                        AssembleReaction(U, R, Rp);
                        for (int i = 0; i < numsubintervals + 1; i++)
                                Gp→add_to_diagonal_entry(i, Rp[i]);
                }

                //Multiply the Matrix A and the vector U
                AU = A→Mult(U);

                //for loop to  create each entry of the vector G
                for (int i = 0; i ≤ numsubintervals; i++)
                {
                        G[i] =  -1*(AU[i] + R[i] - F[i]);
                }

                //solve for h to update U
                //first transform Gp
                Gp→transform();

                //solve the linear system for h
                h = Gp→solve_linear_system(G);

                //Update U
                for (int i = 0; i ≤ numsubintervals; i++)
                {
                        U[i] = U[i] + h[i];
                }

                //delete the Tridiagonal Matrix Gp associated with the iteration
                delete Gp;

                //find the norm of h to see if iterations continue
                norm = find_l2_norm(h);

                //determine if the condition ||U_n+1 - U_n|| < Tolerance has bee
n met
                if (norm < TOL)
                {
                        // if met, break from loop and stop iterations
                        break;
                }

                // update iteration counter
                iteration_counter = iter;
        }

        if (iteration_counter ≡ max_num_iter)
        {
                std::ofstream ofs;
                ofs.open("problem_info.txt", std::ofstream::out | std::ofstream::app
);
                ofs << " Convergence not reached within max number of iterations: " << max_num_it
er << endl;
                ofs.close();
```

```
        }
        else
        {
                std::ofstream ofs;
                ofs.open("problem_info.txt", std::ofstream::out | std::ofstream::app
);
                ofs << " Convergence was reached at iterations = " << iteration_counter << en
dl;
                ofs.close();
        }

        delete A;

        return U;
}



double TwoPointBVPAppr::find_max_error(int max_iters, double TOL)
{
        //generate an approximate solution

        vector<double> approximate_solution = Solve(max_iters, TOL);
        int numberSubintervals = get_numsubintervals();

        // Evaluate the true solution at all the xcoords
        vector<double> true_solution(numberSubintervals);
        vector<double> x(1);

        for (int i = 0; i < numberSubintervals; i++)
        {
                x[0] = xcoord[i];
                true_solution[i] = theproblem→eval_true_solution(x);
        }

        // Compare the true solution to the  approximate
        // solution and store/update the bigest error found
        // durring the sweep.

        // create error at x_i and intialize max error
        double max_error =-1 ;
        double ex_i;

        //for loop to find and update max error
        for (int i = 0; i < numberSubintervals; i++)
        {
                // calculate the current error
                ex_i = fabs(true_solution[i] - approximate_solution[i]);
                // compare the absolute value of the errors
                if (max_error < ex_i)
                {
                        max_error = ex_i;
                }

        }

        return max_error;
}
```

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