HW #2 : Collocation Method

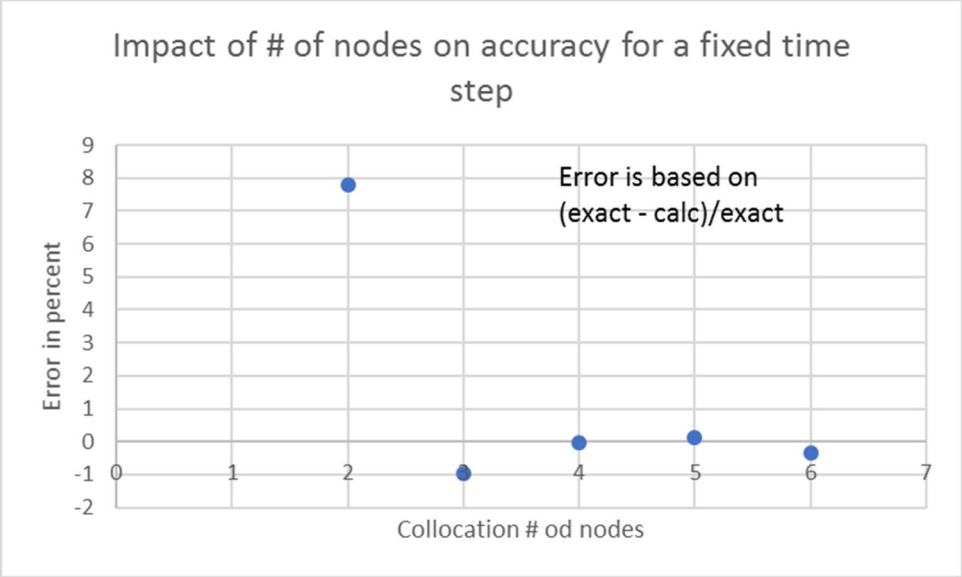
Sylvain Charbonnel

1/18/2018

Results

	2 nodes	3 nodes	4 nodes	5 nodes	6 nodes	dt=0.5, repeat collocation twice with 3 nodes	ODE	Exact solution
X at 1 sec	0.7007	0.7672	0.7602	0.7589	0.7625	To be done at a later time	0.7599	0.7599
Execution time*	35msec	38msec	40msec	41 msec	43 msec	To be done at a later time		N/A

* Execution time is based on doing the simulation three times for each case in order to average out variabilities



Discussion

2 Nodes seems to lack accuracy with the current time step selected. However, a real discussion could occur while selecting 3 nodes vs 4 vs 5 vs 6 nodes. The trade-off is about accuracy vs execution speed which would become relevant under much more involved problem or in case the model would need to run on board in real time with a very fast dynamic plant.

Another consideration for on board implementation is RAM. For simple problems it does not matter from a memory standpoint whether 2 nodes or 6 nodes are used. However, for problem with multiple ODE, memory could become a limiting factor

Conclusion

For computer purposes, with mild complexity, it will be preferred to maybe err on the side of caution and use 5 nodes.

For real time with fast dynamic implementation, 3 nodes is definitely the way to go

Code:

- Platform: C++
- High level, headers

Matrix N_Legendre(int, double);

➔ return the Legendre matrix

⇨ Specify number of nodes used for collocation
⇨ Specify the end time for the collocation

matrix.h

➔ I took it from a website, allows me to inverse a matrix.

double* convert_Matrix_to_ptr(Matrix,int rows, int col);

➔ Convert the matrix into a pointer
⇨ Input the matrix to convert
⇨ Input the number of rows and columns for the array

double * mult_mat(const double* const, const double* const, int, int);

double * mult_mat(const double* const, double, int, int);

➔ Return a pointer which is a multiplication of a vector pointer by another one or by a constant
⇨ Input vector array that needs to be multiplied
⇨ Input either another vector array that multiplies the first one or a constant
⇨ Input number of rows and column

double* funcRHS(const double* const x, double u, double tau, int nodes)

➔ Return a pointer vector, corresponds to the RHS of the equation dx/dt = 1/tau * (f(x))

⇨ Input a pointer vector x, a constant u, a time constant tau, and number of nodes, which is 1 higher than the size of the array

double* func_to_solve_for_zero(const double* const x, const double* const tf_N_inv, double x_init, double u, double tau, int nodes);

➔ Return a pointer vector, which needs to equate 0 to solve the problem accurately. Essentially it is dx/dt – 1/tau*(f(x)). Except that in this case, dx/dt is replaced with its equivalent approximated results based on collocation

⇨ Input a vector x = x(1), x(2), … x(nodes)

⇨ Input t_final * N inversed pointer

⇨ Input initial x

⇨ Input constant u, tau for the equation f(x)

⇨ Input number of nodes for the collocation

Entire Code

```
#include <iostream>

#include <fstream>

#include "math.h"

#include "matrix.h"



using namespace std;



/* PROBLEM STATEMENT

Objective: Solve a differential equation with orthogonal collocation on finite elements.

Create a MATLAB or Python script to simulate and display the results. Estimated Time: 2-3 hours


Solve the following differential equation
```

- 1) from time 0 to 1

- 2) with orthogonal collocation on finite elements with

- 3) 4 nodes for discretization in time.

- 4) 5 dx/dt = -x^2 + u

- 5) Specify the initial condition for x as 0 and

- 6) the value of the input, u, as 4.

Compare the solution result with 2-6 time points (nodes).

Report the solution at the final time for each and comment on how the solution changes with an increase in the number of nodes.

*/

Matrix N_Legendre(int, double);

double* convert_Matrix_to_ptr(Matrix,int rows, int col);

double * mult_mat(const double* const, const double* const, int, int);

double * mult_mat(const double* const, double, int, int);

double* grad_f(const double * const x, unsigned int size_x, const double* const N, double u, double tau, double* (*f)(double*, const Matrix, double, double, int));

double* funcRHS(const double* const x, double u, double tau, int nodes) { // Problem statement # 4

double * func = new double[nodes-1];

for (int i=0;i<nodes-1;i++) { *(func+i) = 1./tau*(-powf(*(x+i),2.0)+u);}

return func;

}

double* func_to_solve_for_zero(const double* const x, const double* const tf_N_inv, double x_init, double u, double tau, int nodes); // Problem statement

```
int main()
{
   const int nodes = 4;      // Set number of nodes, Problem statement #3
   double * x    = new double [nodes-1];

   double x_init = 0.0;                  // Set initial condition, Problem statement # 5
   const double u = 4.0;        // Set input u, Problem statement #6
   const double tau = 5.0;      // parameter for ODE
   const double t_final = 1.0;  // set final time, problem statement # 1

// Transfer ODE into a collocation, Problem statement #2 & 4

   Matrix tn_N = N_Legendre(nodes, t_final); // return the tn x N matrice based on Legendre polynomial assumption, x = A + Bt+Ct^2+Dt^3
                                 // replace dxdt by polynomial and plug the function for RHS to form a new set of algebric equations
   Matrix M= tn_N.inverse();
   cout << M << endl;
 //   cout << M*tn_N<<endl;
   double* tf_N_inv = new double[(nodes-1)*(nodes-1)];
   tf_N_inv = convert_Matrix_to_ptr(M,nodes-1, nodes-1);

// initialize all the x's to the initial condition, and error and iter for the convergence loop
   for (int i=0;i<nodes-1;i++) {*(x+i)=x_init;}
```

```
   double* x_new = new double[nodes-1];
   double long error = 0.0;
   unsigned int iter = 0;

   double long convergence = 2e-6;       // absolute max change acceptable until sto iteration


   do {
      for (int i=0;i<nodes-1;i++) {
         *(x_new+i) = *(x+i) - 0.1* *(func_to_solve_for_zero(x, tf_N_inv,x_init, u, tau, nodes)+i);

          double long error_i = fabs(*(x_new+i)-*(x+i));
         error = (error_i>error) ? error_i:error;
         *(x+i) = *(x_new+i);
      }
     cout <<" error = " << error << "   iter = " << iter << " func to be zero = "
<<*(func_to_solve_for_zero(x, tf_N_inv,x_init, u, tau, nodes)+0)<<endl;
      iter++;
   } while ((error > convergence)&&(iter<50));
   delete [] x_new;

// WRITE RESULTS TO TXT FILE
   ofstream results_file;
   results_file.open("output.txt");
   for (int i=0;i<nodes-1;i++) {results_file << *(x+i)<<endl;}
   results_file.close();
```

```
   cout << "final residuals" << endl;
      for (int i=0;i<nodes-1;i++) {cout << "func = " <<*(func_to_solve_for_zero(x, tf_N_inv,x_init, u, tau,
nodes)+i)<<endl;}
   cout << "final answer for x " << endl;
      for (int i=0;i<nodes-1;i++) { cout <<"x[" << i+1 <<"] = " << *(x+i) <<endl;}
   delete [] x;
return 0;
}

double * mult_mat(const double* const M, const double* const Vec, int rows, int col) {
   double* M3 = new double[rows];
   for (int i=0;i<rows;i++) {
         *(M3+i) =0;
         for (int j=0;j<col;j++) {
            *(M3+i) = *(M3+i) + *(M+j+i*col) * *(Vec+j);
         }
   }

return M3;
}

double * mult_mat(const double* const M, double alpha, int rows, int col) {
   double* M3 = new double[rows*col];
   for (int i=0;i<rows;i++) {
         for (int j=0;j<col;j++) {
            *(M3+j+i*col) = *(M+j+i*col)* alpha;
         }
   }
```

```
return M3;
}

double* convert_Matrix_to_ptr(Matrix M,int rows, int col) {
   double * vec = new double[rows*col];
   for (int i=0;i<rows;i++) {
      for (int j=0;j<col;j++) {
         *(vec+j+i*col) = M(i,j);
      }
   }
   return vec;
}


Matrix N_Legendre(int nodes, double t_end) {

      switch (nodes-1) {
      case 1:
         { Matrix N(1,1);
         N(0,0) = 1.0;
         N*=t_end;
         cout <<"N in Legendre = " << endl <<N << endl;
         return N;
         break;}
      case 2:
         { Matrix N(2,2);
         N(0,0) = 0.75; N(0,1) = -0.25;
         N(1,0) = 1.00; N(1,1) =  0.00;
         N*=t_end;
```

```
    cout <<"N in Legendre = " << endl <<N << endl;
    return N;
    break;}
case 3:
    {Matrix N(3,3);
    N(0,0) = 0.436; N(0,1) = -0.281; N(0,2) = 0.121;
    N(1,0) = 0.614; N(1,1) =  0.064; N(1,2) = 0.046;
    N(2,0) = 0.603; N(2,1) =  0.230; N(2,2) = 0.167;
    N*=t_end;

    cout <<"N in Legendre = " << endl <<N << endl;
    return N;
    break;}
case 4:
    {Matrix N(4,4);
    N(0,0) = 0.278; N(0,1) = -0.202; N(0,2) = 0.169; N(0,3)  = -0.071;
    N(1,0) = 0.398; N(1,1) =  0.069; N(1,2) = 0.064; N(1,3)  = -0.031;
    N(2,0) = 0.387; N(2,1) =  0.234; N(2,2) = 0.278; N(2,3) = -0.071;
    N(3,0) = 0.389; N(3,1) =  0.222; N(3,2) = 0.389; N(3,3) =  0.000;
    N*=t_end;

    cout <<"N in Legendre = " << endl <<N << endl;
    return N;
    break;}
case 5:
    {Matrix N(5,5);
    N(0,0) = 0.191; N(0,1) = -0.147; N(0,2) = 0.139; N(0,3) = -0.113; N(0,4) = 0.047;
    N(1,0) = 0.276; N(1,1) =  0.059; N(1,2) = 0.051; N(1,3) = -0.050; N(1,4) = 0.022;
```

```
        N(2,0) = 0.267; N(2,1) =  0.193; N(2,2) = 0.251; N(2,3) = -0.114; N(2,4) = 0.045;
        N(3,0) = 0.269; N(3,1) =  0.178; N(3,2) = 0.384; N(3,3) =  0.032; N(3,4) = 0.019;
        N(4,0) = 0.269; N(4,1) =  0.181; N(4,2) = 0.374; N(4,3) =  0.110; N(4,4) = 0.067;
        N*=t_end;

        cout <<"N in Legendre = " << endl <<N << endl;
        return N;
        break;}
      default:
        cout << "number of nodes lower or higher than has been coded, nodes >=2, <=6" << endl;
   }
}

double* func_to_solve_for_zero(const double* const x, const double* const tf_N_inv, double x_init,
double u, double tau, int nodes) { // Problem statement # 4
   double * func = new double[nodes-1];
   double *v_x_init = new double [nodes-1];
   for (int i = 0;i<nodes-1;i++) {*(v_x_init+i) = x_init;}
   for (int i=0;i<nodes-1;i++) { *(func+i) = *(mult_mat(tf_N_inv, x, nodes-1, nodes-1)+i) -
*(mult_mat(tf_N_inv, v_x_init, nodes-1, nodes-1)+i) - *(funcRHS(x,u,tau,nodes)+i);}
return func;
}

/*
double* grad_f(const double * const x, unsigned int size_x, const double* const N, double u, double tau,
double* (*f)(const double* const, const Matrix, double, double, int)) {
    double delta_x = 0.01;
    double * dx = new double[size_x];
```

```
    double *grad = new double[size_x];
    double *x_plus_delta = new double[size_x];
    double *x_minus_delta = new double[size_x];
    for (unsigned int i=0; i<size_x;i++) {
        *(x_plus_delta+i) = *(x+i);
        *(x_minus_delta+i) = *(x+i);
        *(dx+i) = *(x+i) * delta_x;
    }

    for (unsigned int i=0; i<size_x;i++) {

            *(x_plus_delta+i)=*(x+i)+ *(dx+i);
            *(x_minus_delta+i)=*(x+i)- *(dx+i);
            *(grad+i) = (*f(x_plus_delta, N,u,tau,size_x)- *f(x_minus_delta, N,u,tau,size_x))/(2.0* *(dx+i));
            *(x_plus_delta+i)=*(x+i);
            *(x_minus_delta+i)=*(x+i);
    }
    delete [] x_plus_delta;
    delete [] x_minus_delta;
    delete [] dx;
return grad;
}

*/
```