ME257 - Homework 2

Submitted by : Saurabh Sharma

Question 1

(i)

`typename _Scalar:` Indicates the data type of the scalar coefficients (int, float, double etc.). The data type of the values stored in the sparse matrix will be of the type `_Scalar.`

`int _Options:` Controls the way in which the matrix is stored i.e either in `RowMajor` or in `ColMajor` format. Integer value of 0 corresponds to `ColMajor` format while value of 1 corresponds to `RowMajor` format. By default, `int _Options:` is set to 0 which is a `ColMajor` storage format.

`typename _StorageIndex:` Indicates the data type of the indices used for the sparse matrix. The entries of the sparse matrix can be accessed through these indices and they must be of a signed data type (short, int). The default data type for this parameter is `int.`

(ii)

(a) The keyword `typedef` creates an alias name for the class being created. In this example a triplet class is created for a certain choice of parameters. `SampleSpMatClass` can be used to create new classes of the particular triplet. For example consider the below code, where new classes `MassMatrix` and `StiffnessMatrix` are created and they are of the same type as `SampleSpMatClass.`. These `typedef` is particularly useful while defined function taking class objects as arguments. Explicitly defining the class object through the template format for every function can be tedious and using `typedef` option one can make the code more readable.

```
SampleSpMatClass MassMat, StiffnessMatrix;
```

(b) `Eigen::SparseMatrix <int, 0, int>`

(c) `Eigen::SparseMatrix <double, 1, int>`

(iii)

(a) In the class template `Eigen::SparseMatrtix<_Scalar, _Options, _StorageIndex>`, the parameters `_Options` and `_StorageIndex` are optional. These parameters have default values which are 0 for `_Options` and data type of `int`. If the user specifies these parameters, the default values are overridden. In the case of user not providing values for these parameters, the class is created using the predefined default values. In the given example, `A` is an object of the class, storing 1000×1000 entries of `double` values in `ColMajor` format and the entries of the sparse matrix can be accessed by integer indices.

(b) Populating of the sparse matrix was tried using both Method 1 and Method 2 for different values of of `N`. Table 1 summarises the run time for each of the methods under different values of `N`. It can be clearly seen that Method2 out performs Method1 under for all values of `N`, however as the order of magnitude of `N` increases, the difference in run-times between the methods is very significant.

Table 1: Run time comparison of the methods

Case	`N`	Method1 runtime / ms	Method2 runtime / ms
1	1×10^3	27.0	2.0
2	1×10^4	1000.0	22.0
3	1×10^5	117115.0	95.0

The slower runtime of method1 is due to the time taken to access the specified memory location make the necessary value assignment operation. The sparse matrix is by default stored in the column major format unless otherwise specified. The memory allocated is like a single long block of memory where the matrix is stored column by column. When accessing adjacent column elements, the pointer needs to jump over a large memory location to get to the correct entry. This back and forth movement between the memory increases the access time and results in longer runtime. As the size of `N` increases, the memory access time becomes the bottle neck for faster code execution. Also method1 allocates $N \times N$ memory cells, but given the nature of the sparse matrix system, most of the entries are zero. The memory allocated for the zero entries can be removed which results in better memory management thus improving the runtime. In method2, only `3N` memory locations are requested. Also, in method2, fewer memory accesses operations are needed to fully populate the matrix.

The method `reserve()` aids in reserving a chuck of memory based on the user input. Every time an entry is appended into the vector, a certain time is elapsed in allocation of the memory. On using the method `reserve()`, the compiler already knows the allocated reserved memory location and appends the entry directly into these memory locations. This makes the appending operation computationally more efficient.

The method `shrink_to_fit()` removes any unused memory which was reserved for the vector. If the actual size of the vector is `N` and `M` memory units were originally reserved, then `(M-N)` memories locations are unused by the vector. The method `shrink_to_fit()` trims the memory allocated to exactly to the memory used, thereby freeing up memory space, which can in turn be used by other functions resulting in improved performance.

Question 2

The source code is attached with this report,

The basis function $N_1(x)$ and $N_2(x)$ are defined as follows, where X_a and X_b are the left and right nodal coordinates of the element/interval in consideration.

$$N_1(x) = (X_b - x)/(X_b - X_a)$$

$$N_2(x) = (x - X_a)/(X_b - X_a)$$

All the runs are performed with a default of ten nodes `(nNode = 10)`. Five different functions are used in this analysis. The different functions can be selected by setting the int parameter `FunctionIdentifier` to 1,2,3,4 or 5. Table 2 lists down the various function used in this analysis. The functions chosen are well behaved in $[0, 1]$ and belong to $L_2[0, 1]$. A mix of functions like polynomial, exponential and trigonometric functions are chosen to test the code.

Table 2: Functions corresponding to the user input

`FunctionIdentifier`	$f(x)$
1	$sin(\pi x)$
2	x^2
3	e^x
4	$sin^2(3x)$
5	$cos(3\pi x)$

(i)

The array `double NodalCoordintates[nNodes]` contains the coordinates of all the nodes. The function `getCorodinates(NodalCoordinates, h, nNodes)` is used to compute the coordinates and populate the array.

(ii)

The array `int Connectivity[2*nElements]` contains the local to global map for the elements. The function `getConnectivity(Connectivity, nElements)` populate the array.

(iii)

A function `ComputeElementForceVector(FunctionIdentifier,Xa,Xb, ElemForceVector)` is used to compute the element force vector of size 2 and to store the values in `double ElemForceVector[]`. The element force vector is defined as follows.

$$\texttt{ElemForceVector[0]} = \int_{X_a}^{X_b} f(x)N_1(x)dx$$

$$\texttt{ElemForceVector[1]} = \int_{X_a}^{X_b} f(x)N_2(x)dx$$

The element force vector depends of the function $f(x)$ and has to be evaluated for each choice of $f(xs)$. The tool **MATHEMATICA** was used to evaluate the closed form of the integrals described above. The closed form relations for `ElemForceVector[0]` and `ElemForceVector[1]` for different choices of functions are indicated in table 3.

Table 3: Closed form expressions for different choice of functions

$f(x)$	`ElemForceVector[0]`	`ElemForceVector[1]`
$sin(\pi x)$	$\frac{\pi(X_a-X_b)cos(\pi X_a)-sin(\pi X_a)+sin(\pi X_b)}{\pi^2(X_a-X_b)}$	$\frac{\pi(-X_a+X_b)cos(\pi X_b)+sin(\pi X_a)-sin(\pi X_b)}{\pi^2(X_a-X_b)}$
x^2	$\frac{-(X_a-X_b)(3X_a^2+2X_aX_b+X_b^2)}{12}$	$\frac{-(X_a-X_b)(X_a^2+2X_aX_b+3X_b^2)}{12}$
e^x	$\frac{-e^{X_b}+e^{X_a}(1-X_a+X_b)}{X_a-X_b}$	$\frac{-e^{X_a}+e^{X_b}(1+X_a-X_b)}{X_a-X_b}$
$sin^2(3x)$	$\frac{cos(6X_a)-cos(6X_b)+6(X_a-X_b)(-3X_a+3X_b+sin(6X_a))}{72(X_a-X_b)}$	$\frac{-cos(6X_a)+cos(6X_b)-6(X_a-X_b)(3X_a-3X_b+sin(6X_b))}{72(X_a-X_b)}$
$cos(3\pi x)$	$\frac{-cos(3\pi X_a)+cos(3\pi X_b)+3\pi(-X_a+X_b)sin(3\pi X_a)}{9\pi^2(X_a-X_b)}$	$\frac{cos(3\pi X_a)-cos(3\pi X_b)+3\pi(X_a-X_b)sin(3\pi X_b)}{9\pi^2(X_a-X_b)}$

(iv)
A function `ComputeElementMassMatrix(Xa,Xb,ElemMassMatrix)` is used to compute the element mass matrix of size 2×2 and values are stored in `double ElemMassMatrix[2][2]`. The element mass matrix is defined as follows and the close form expression for the integral is obtained using **MATHEMATICA**.

$$\texttt{ElementMassMatrix[0][0]} = \int_{X_a}^{X_b} N_1(x)N_1(x)dx = (X_b - X_a)/3$$

$$\texttt{ElementMassMatrix[0][1]} = \int_{X_a}^{X_b} N_1(x)N_2(x)dx = (X_b - X_a)/6$$

$$\texttt{ElementMassMatrix[1][0]} = \int_{X_a}^{X_b} N_2(x)N_1(x)dx = (X_b - X_a)/6$$

$$\texttt{ElementMassMatrix[1][1]} = \int_{X_a}^{X_b} N_2(x)N_2(x)dx = (X_b - X_a)/3$$

The element mass matrix does not depend on the function and for equal interval lengths, the elemental mass matrix is identical for all the elements.

(v)
A function `ComputeGlobalMassMatrix (&matTriplets, &GlobalMassMatrix, NodalCoordinates, Connectivit, nElements)` is used to compute the global mass matrix. The parameter `matTriplets` and `GlobalMassMatrix` are passed by reference to function to ensure the changes made to the variables within the function is reflected in the main function. The function to compute the element stiffness matrix

`ComputeElementMassMatrix` is called from within `ComputeGlobalMassMatrix()`.

A function `ComputeGlobalForceVector (FunctionIdentifier, &ForceVec, nElements, NodalCoordinates, Connectivity` is used to compute the global force vector `ForceVec`. The parameter `ForceVec` is passed by reference to function to ensure the changes made to the variables within the function is reflected in the main function. The function to compute the element force vector `ComputeElementForceVector` is called from within `ComputeGlobalForceVector()`. Based on the value of `FunctionIdentifier`, the function `ComputeElementForceVector` returns the appropriate element force vectors which gets assembled into `ForceVec` .

(vi)
A sparse LU solver object `Solver` is created and the resulting system of linear equations was solved.

(vii & viii)
The output of the program is written to file *solution.dat*, which contains the nodal coordinates and corresponding DOF values. Figure 1,2,3,4 and 5 show the comparison of $f(x)$ and $P_h f(x)$ for different functions $f(x)$. Note that the results indicated are for `nNodes = 10.` All the functions chosen in the exercise belongs to $L_2[0,1]$. The function $f(x) = sin^2(3x)$ was chosen because, the function has curvature changes within $[0,1]$ and the objective was to understand how good the L2 projection captures these curvature changes. It is seen from figure 4 that with 10 nodes, the approximation curve is close to the actual function. It is clearly seen in figure 5 that using 10 nodes, the approximation $(P_h f)(x)$ is unable to accurately capture curve $f(x) = cos(3\pi x)$. This is due to large changes in curvature $(f''(x))$, which the piecewise linear approximation is unable to capture well with fewer nodes.

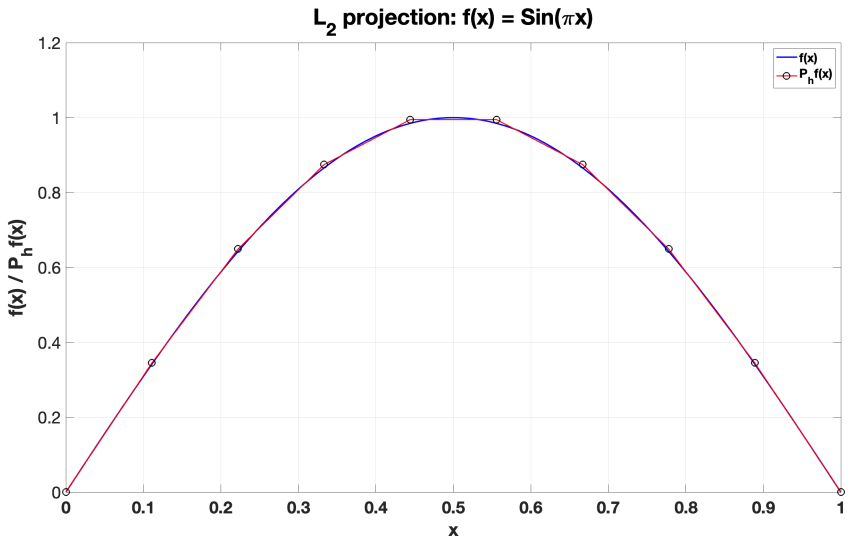


Figure 1: Comparison of $f(x)$ and $(P_h f)(x)$ for $f(x) = sin(\pi x)$

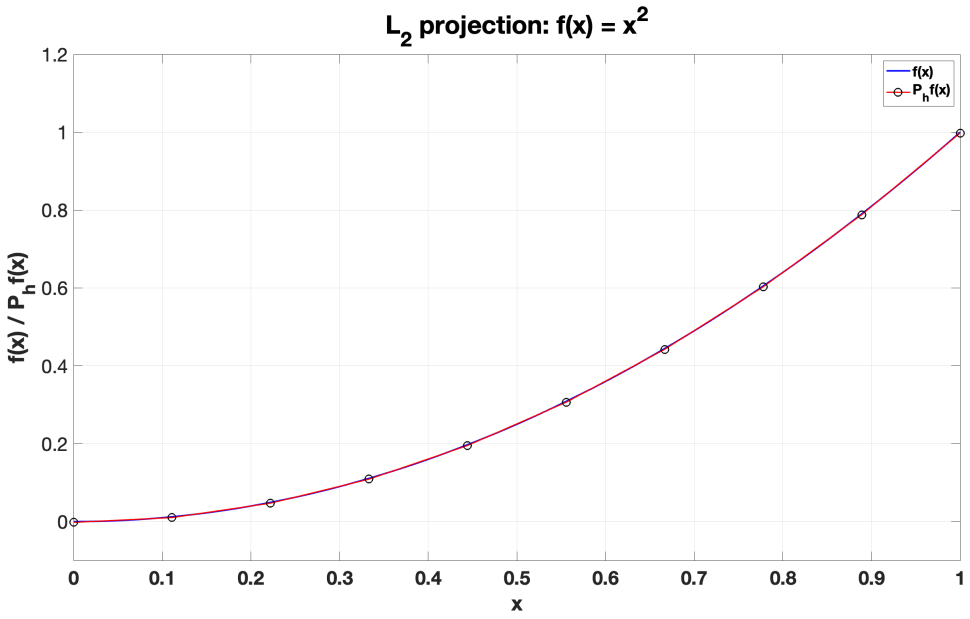


Figure 2: Comparison of $f(x)$ and $(P_h f)(x)$ for $f(x) = x^2$

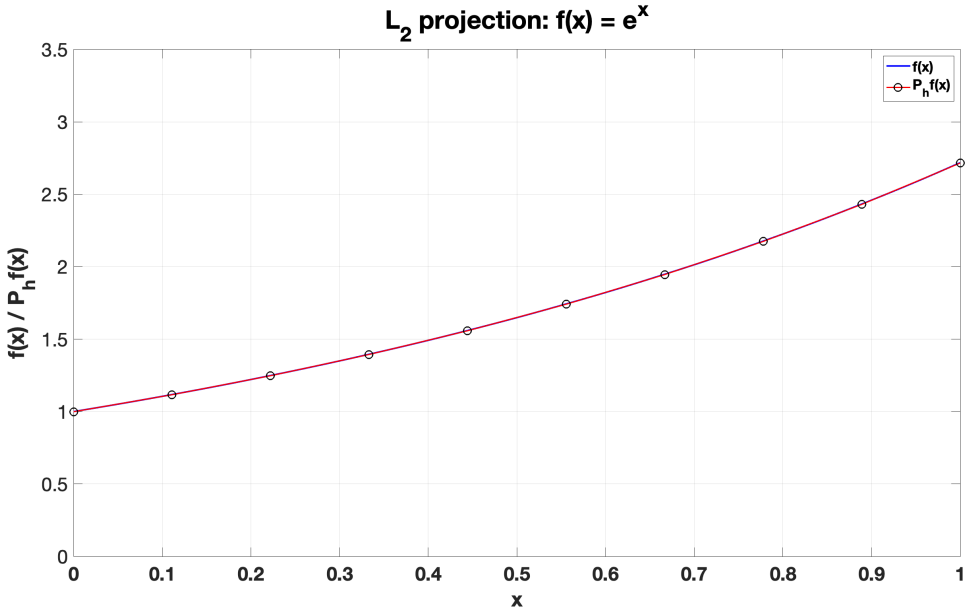


Figure 3: Comparison of $f(x)$ and $(P_h f)(x)$ for $f(x) = e^x$

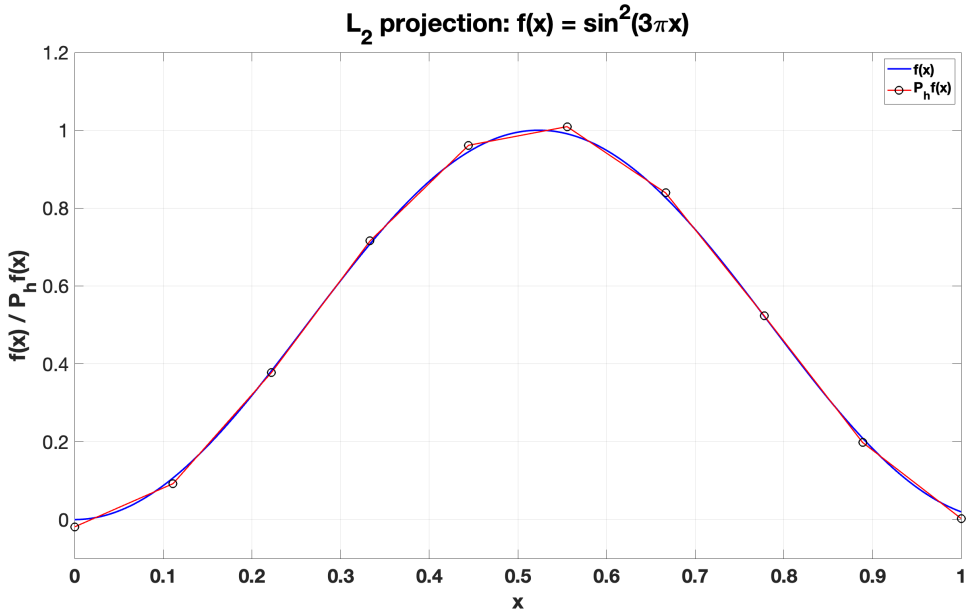


Figure 4: Comparison of $f(x)$ and $(P_h)f(x)$ for $f(x) = sin^2(3x)$

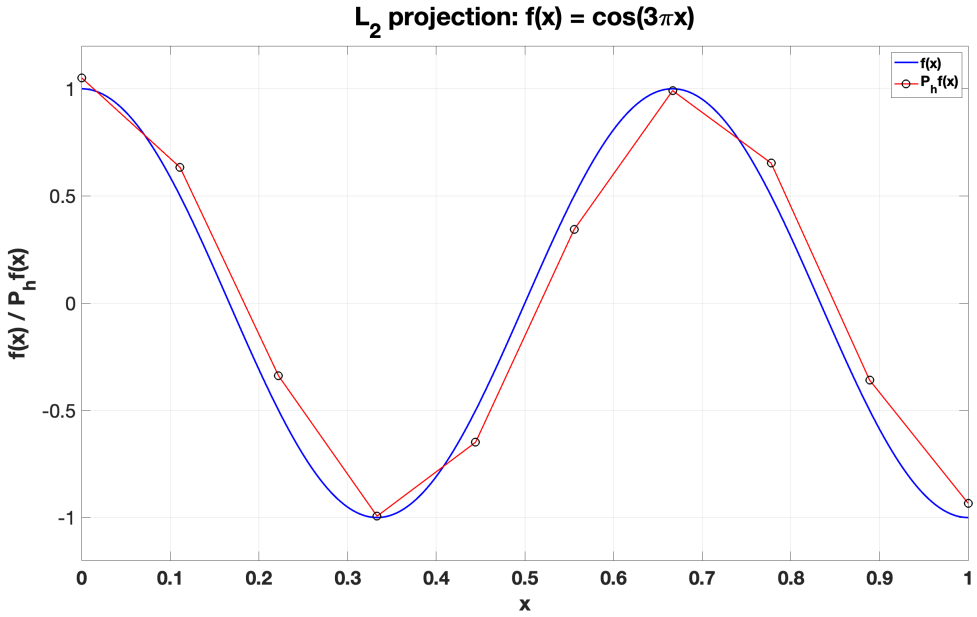
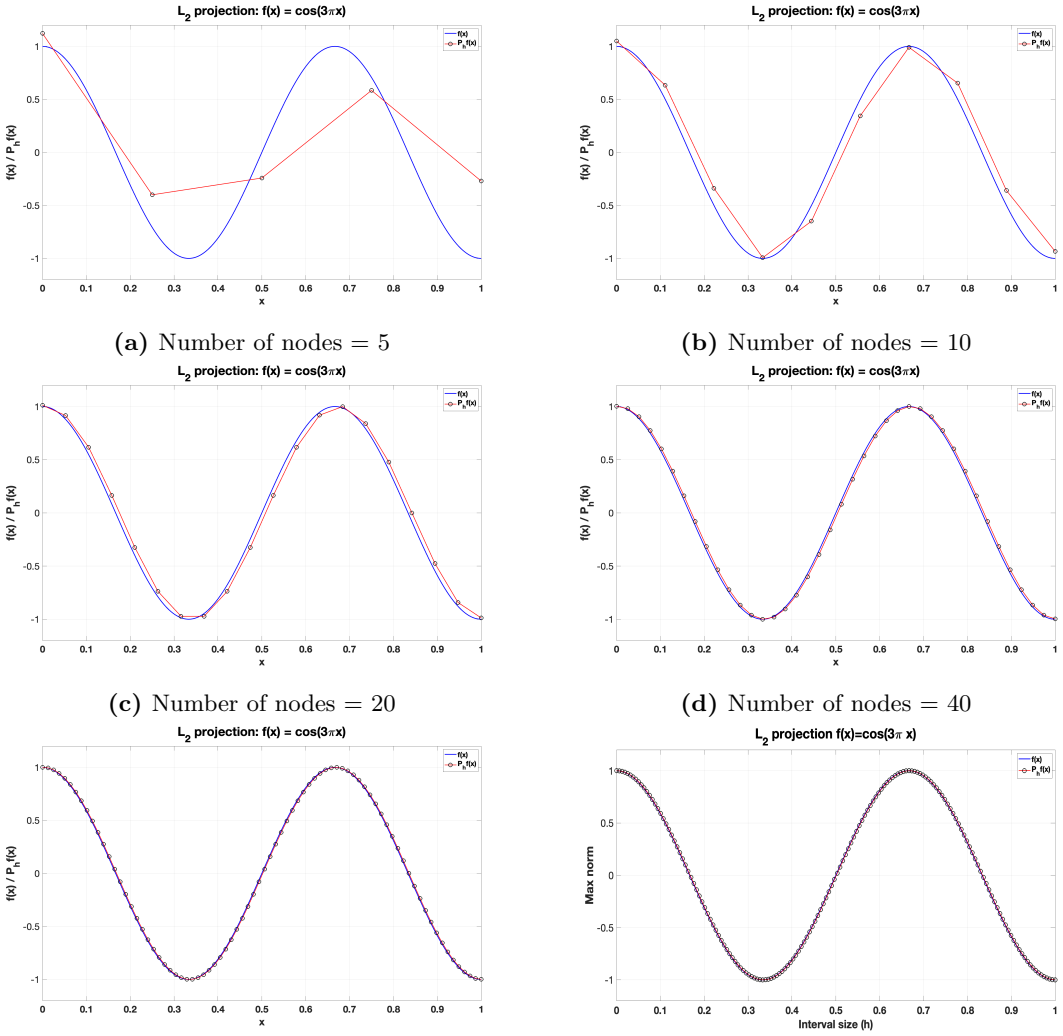


Figure 5: Comparison of $f(x)$ and $(P_h f)(x)$ for $f(x) = cos(3\pi x)$

To better approximate the function $cos(3\pi x)$, a refinement of the interval size was done. The simulation was carried out with different number of nodes (5,10,20,40,80,160) and the $L2$ projection for different number of grid points is shown in figure 6. As expected, with increase in the number of nodes, the L2 projection tends closer to the analytical solution. With more nodal values available, the piece wise linear approximation is able to capture the curvature changes in a better way.



(a) Number of nodes = 5

(b) Number of nodes = 10

(c) Number of nodes = 20

(d) Number of nodes = 40

(e) Number of nodes = 80

(f) Number of nodes = 160

Figure 6: Behaviour of $P_h f(x)$ with increase in number of nodes

(ix)
The error is estimated using the $MaxNorm$ which is the absolute maximum value of the difference in the function value at nodal points and the approximated value at the same nodal point. A function `getMaxNorm (nNodes, NodalCoordinates, DOFSolution, FunctionIdentifier, Error)` is defined which calculate the absolute error between $f(x)$ and $(P_h f)(x)$ at each nodal point and returns the maximum of these errors. The MaxNsorm can be defined as follows, where nNodes is the number of nodes.

$$MaxNorm = max\{f(x_i) - (P_h f)(x_i)\} \quad \forall\ i\ \in\ [0, 1, 2...nNodes - 1]$$

In this exercise, refinement in number of nodes is done for the function $cos(3\pi x)$. Table 4 gives the values of the MaxNorm for different number of nodes while table 5 gives the MaxNorm for different interval sizes h.

Table 4: MaxNorm for different number of nodes

nNodes	MaxNorm
5	0.729856
10	0.160836
20	0.082497
40	0.040025
80	0.019884
160	0.00987546

Table 5: MaxNorm for different interval sizes

Interval size h	MaxNorm
0.25	0.729856
0.1111111	0.160836
0.0526316	0.082497
0.0256410	0.040025
0.0126582	0.019884
0.0062893	0.00987546

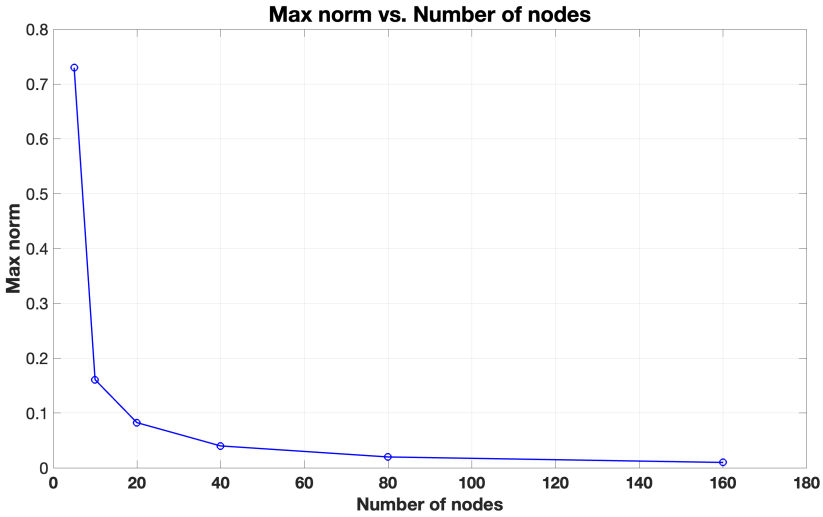


Figure 7: Variation of MaxNorm as a function of number of nodes

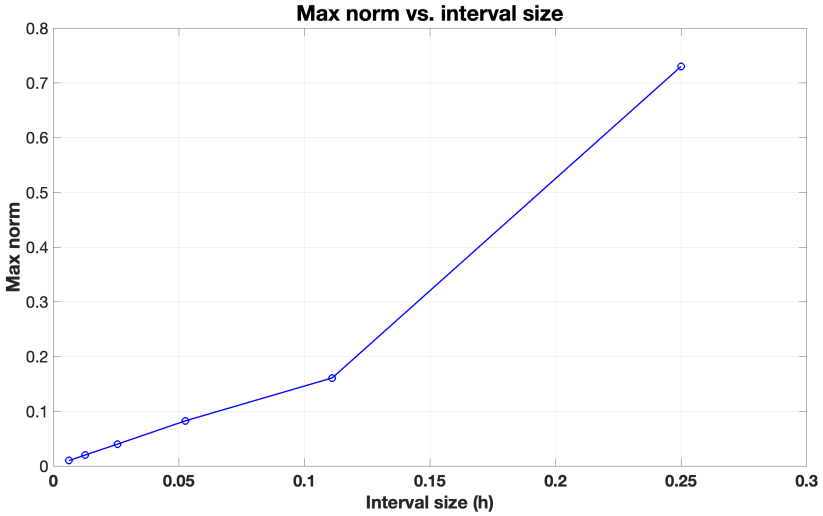


Figure 8: Variation of MaxNorm as a function of interval size

It can be seen from figure 7 that the error decreases as the number of nodes increase, which is the expected result. Figure 8, shows the variation of the error as a function of the interval length h, which indicates that the error reduces as the interval size reduces.

——— o ——— o ———