UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Civil, Arquitetura e Urbanismo

IC639: Métodos Numéricos para Engenharia Civil

List 7

IVB, BVP, Least Square Method

Student:

Carlos Henrique Chama Puga - 195416

Advisors:

Porf. Dr. Philippe Devloo

Dr. Giovane Avancini

Campinas

2024

Contents

1 Introduction

The final list of the course on Numerical Methods in Civil Engineering is divided into three main topics: Initial Value Problem (IVP), Boundary Value Problem (BVP), and Least Square Method. All three topics are better discussed in the following sections. During the resolution of the exercises, the following bibliography is used for further understanding of the topics: (1, 2).

1.1 Initial Value Problem (IVP)

In science and engineering, many problems can be modeled using differential equations. In these problems, the rate of change of one or more variables with respect to another (space or time, for example) is considered.

In the majority of cases, these differential equations are not easily solved analytically so numerical methods are used to approximate the solution. Initial Value Problems are a type of differential equation where the solution is known at a single point. The solution is then propagated to other points using numerical methods such as the Runge-Kutta method.

Runge-Kutta methods, in general, are a family of numerical methods used to solve ordinary differential equations. The biggest advantage of this approach, when compared to Taylor's methods, is that they do not require the computation of the derivatives of the function, which is computationally expensive. The most common Runge-Kutta method employed is the second- and fourth-order methods, which can be expressed through the Butcher Tableau.

Let Eq. (1.1) be the ordinary differential equation to be solved

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0, \tag{1.1}$$

then, the Runge-Kutta methods take the form of

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i, \tag{1.2}$$

in which h is the step size, s is the number of stages, b_i are the weights (found in the Butcher Tableau), and k_i are the intermediate values.

The intermediate values are calculated as

$$k_i = f(x_n + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j), \tag{1.3}$$

where c_i and a_{ij} are the coefficients of the Butcher Tableau. A given Butcher Tableau is given of the form

$$\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}$$

Exercise 1 implements a class capable of reading a given Butcher Tableau and performing the Runge-Kutta method to solve the equation

$$y' = 1 + (x - y)^2, \quad \forall 2 \leq x \leq 3, \quad y(2) = 1, \tag{1.4}$$

using the following Runge-Kutta method:

$$\begin{array}{c|cccc}
0 & & & & \\
1/3 & 1/3 & & & \\
2/3 & -1/3 & 1 & & \\
1 & 1 & -1 & 1 & \\
\hline
 & 1/8 & 3/8 & 3/8 & 1/8
\end{array}$$

1.2 Boundary Value Problem (BVP)

In the previous section, the concept of Initial Value Problems is discussed. In this type of problem, all the specified conditions are given at a single point, usually at the beginning of the interval. For physical problems that are position-dependent rather than time-dependent, the Boundary Value Problem (BVP) is more appropriate, as the conditions are given at more than one point.

Among the most common methods for solving Boundary Value Problems are the Finite Difference Method and the Finite Element Method. In this list, the ideas of the Galerkin Method, widely used in the Finite Element Method, are used to solve the following BVP

$$\begin{cases} -\nabla \cdot \nabla u(x,y) = f(x,y) & \text{in } \Omega, \\ u(x,y) = u_D & \text{on } \partial\Omega_D, \\ \nabla u(x,y) \cdot \boldsymbol{n} = h & \text{on } \partial\Omega_N, \end{cases} \tag{1.5}$$

where Ω is the domain in which the differential equation is defined (in this case the square $[-1,1] \times [-1,1]$), $\partial\Omega_D$ is the boundary where the Dirichlet condition is applied (left and bottom), $\partial\Omega_N$ is the boundary where the Neumann condition is applied (right and top), u_D is the Dirichlet condition, h is the Neumann condition, and $f(x,y)$ is the source term such as the analytical solution is

$$u_{ex} = (x+1)(x-1)\arctan(x-1)\arctan(y-1). \tag{1.6}$$

The first step to solving the BVP using the Galerkin Method is to find the weak formulation for the problem. In this case, the weak formulation is given by

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\partial\Omega_N} h v \, ds, \quad \forall v \in V, \tag{1.7}$$

in which V is the space of test functions and v is the test function.

The second step, and the Galerkin method itself, is to approximate functions u and v by a finite set of basis functions so that

$$u(x,y) \approx u_h = \sum_{i=1}^{n} \alpha_i \phi_i(x,y), \quad v(x,y) \approx \sum_{i=1}^{n} \beta_i \phi_i(x,y). \tag{1.8}$$

where α_i and β_i are the coefficients of the approximation, $\phi_i(x,y)$ are the basis functions, and n is the number of basis functions.

During the resolution of the problem, Legendre polynomials are used as the basis functions for the approximation space. The order of the polynomial is set $k = \{1,2,3,4,5\}$ and the results are compared. The weak formulation becomes

$$\sum_{i=1}^{n} \alpha_i \int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, dx = \sum_{i=1}^{n} \left(\int_\Omega f \phi_j \, dx + \int_{\partial\Omega_N} h \phi_j \, ds \right), \tag{1.9}$$

By calling K the stiffness matrix and F the load vector, the system of equations to be solved is given by

$$K\boldsymbol{\alpha} = F, \tag{1.10}$$

where

$$K_{ij} = \int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, dx, \tag{1.11}$$

$$F_i = \int_\Omega f \phi_i \, dx + \int_{\partial\Omega_N} h \phi_i \, ds. \tag{1.12}$$

As a way to compare the approximation with the analytical solution, the L^2

norm is used so that the approximation error is calculated by

$$\|e\| = \sqrt{\int_\Omega (u_{ex} - u_h)^2 \, dx}. \tag{1.13}$$

1.3 Least Square Method

The Least Square Method is used to approximate a set of data points to a given function. This new function now interpolates nontabulated points so that the error between the data points and the function is minimized.

The method is based on minimizing the sum of the squares of the residuals, which are the differences between the observed values and the values predicted by the model. Least Squares Methods are widely used in regression analysis, curve fitting, and solving overdetermined systems of equations.

Let E be the error function given by

$$E = \sum_{i=1}^{m} (y_i - P_n(x_i))^2, \tag{1.14}$$

where m is the number of data points, n is the degree of the polynomial, a_j are the coefficients of the polynomial, x_i and y_i are the data points, and $P_n(x_i)$ is the polynomial evaluated at x_i given by

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n. \tag{1.15}$$

The error function is minimized by solving the linear system of equations associated with the derivatives of the error function with respect to the coefficients a_{ij} of the polynomial

$$\frac{\partial E}{\partial a_j} = 0 \to \sum_{k=0}^{n} a_k \sum_{i=1}^{m} x_i^{j+k} = \sum_{i=1}^{m} y_i x_i^j, \quad \forall j = 0, 1, \ldots, n. \tag{1.16}$$

By solving the system of equations in Eq. (1.16) the coefficients of the polynomial are found and the function that best fits the data points is obtained. However, the data points may not be perfectly interpolated by the polynomial. In this case, an exponential function can be used to fit the data points

$$y_i = b x_i^a. \tag{1.17}$$

When the data points are fit by Eq. (1.17), two approaches can be taken: 1) linearize the equation by taking the logarithm of the data points or 2) use a nonlinear solver to find the coefficients a and b that minimize the error function.

In 1), the following procedure is employed: eq. (1.17) is transformed into

$$\ln(y_i) = \ln(b) + a\ln(x_i), \tag{1.18}$$

and the error function is minimized by solving the linear system as if it were a first-degree polynomial. Then the coefficients a and b are found by taking the exponential of the coefficients found in the linear system.

In 2), the error function is given by

$$E_{exp} = \sum_{i=1}^{m} (y_i - b x_i^a)^2, \tag{1.19}$$

and the nonlinear system is formed by the derivatives of Eq. (1.19) with respect to the coefficients a and b

$$\frac{\partial E_{exp}}{\partial a} = 2\sum_{i=1}^{m} (y_i - b x_i^a)(-b\ln(x_i)x_i^a) = 0, \tag{1.20}$$

$$\frac{\partial E_{exp}}{\partial b} = 2\sum_{i=1}^{m} (y_i - b x_i^a)(-x_i^a) = 0. \tag{1.21}$$

Solving the nonlinear system of equations formed by eqs. (1.20) and (1.21) leads to the coefficients a and b that minimize the error function. To solve the system, any method can be used, such as the Newton method, the Broyden method, or the Secant method.

It is important to mention that solving the linearization of the exponential function, although computationally less expensive, does not return the Least Squares solution. In other words, the solution of the linearization is expected to be less accurate than the solution of the nonlinear system of equations.

For exercise 3, a set of data points is given, and a code must be implemented so that the data points are fit by an exponential function (linearized and nonlinearized). The results are then compared and the error between the both approximates is calculated.

2 Code Implementation

This section presents the implementation of the classes used to solve the exercises proposed. For solving IVP, the RungeKutta class is developed. For BVP, the Galerkin class is implemented. Finally, for the Least Square Method, the LeastSquare class is created.

Code 1 shows the main function for the Runge-Kutta problem.

```python
def RungeKuttaMethod()->None:
    ft = lambda t, y: 1 + (t - y) ** 2

    rk = RungeKutta(ft, 2, 3, 0.1, 1)

    # 3/8 rule 4th order Runge-Kutta method
    rk.SetButcherTableau(method = "ThreeEights")
    rk.Run()

    print(rk.sol)
```

```python
11
12      rk.WriteResults("List7/Results_RK.txt")
```

Code Listing 1: Main function for the Runge-Kutta problem

On line 2, the right-hand side of the differential equation is defined. On line 4, the RungeKutta object is created with the function, initial and final times, time step, and initial condition. The Butcher Tableau is set on line 7, and the Run method is called on line 8. The results are printed on line 10, and written to a file on line 12.

The main function for the Galerkin Method is shown in Code 2

```python
def GalerkinMethod()->None:
    uex = "exact solution"

    dudx = "exact derivative x"
    dudy = "exact derivative y"

    b = "source term -> negative of the laplacian of the exact solution"

    g = Galerkin(uex, b, dudx, dudy, 5)

    g.Run()

    print(g.alpha)
    print(g.error)
```

Code Listing 2: Main function for the Galerkin Method

The main function for the Least Squares Method is shown in Code 3

```python
def LeastSquareMethod()->None:
    data = [xi, yi]

    squares = LeastSquare(*zip(*data), "NonLinear")
    squares.Run()

    print(f"{squares.alpha=}")
    print(f"{squares.approx_solution=}")
```

```
 9      print(f"{squares.errors=}")
10      print(f"{squares.total_error=}")
```

Code Listing 3: Main function for the Least Squares Method

In which line 2 sets the exact solution (in this case a lambda function can be used). Lines 4 and 5 set the exact derivative of u_{ex} in x and y, respectively. Line 7 sets the source term, which in this case is the negative of the laplacian of the exact solution. The Galerkin object is created on line 9, and the Run method is called on line 11. The results are printed on lines 13 and 14.

Where line 2 defines the set of data points. The LeastSquare object is created on line 4, and the run method is called on line 5. Note that in this example, a non-linear approximation is set. The results are printed on lines 7 to 10.

2.1 The RungeKutta Class

The RungeKutta class' attributes are displayed in Code 4.

```
@dataclass
class RungeKutta:
    ft: callable
    t0: float
    tf: float
    Dt: float
    y0: float

    ButcherTableau: list[float] = field(init=False, default_factory=list)
    c: list[float] = field(init=False, default_factory=list)
    a: list[list[float]] = field(init=False, default_factory=list)
    b: list[float] = field(init=False, default_factory=list)

    sol: list[float] = field(init=False, default_factory=list)
    step: float = field(init=False, default=0)
```

Code Listing 4: Attributes of the RungeKutta class

Where f_t is the function to be solved, t0 and tf are the initial and final times, Dt is the time step, and y0 is the initial condition. The Butcher Tableau is a list of lists containing the coefficients of the Runge-Kutta method. The c, a, and b lists are the coefficients of the Butcher Tableau. The sol list stores the solution to the problem, and the step attribute is used to store the current time step.

As methods, the class has the SetButcherTableau, ConstantK, and Run. Other methods such as the EulerMethod, RK2, RK4, ThreeEightsRule, and WriteResults are also implemented.

The SetButcherTableau method is shown in Code 5

```python
def SetButcherTableau(self, **var)->None:
    butcher = {"euler": self.EulerMethod(), "rk2": self.RK2(), "rk4": self.
   RK4(), "ThreeEights": self.ThreeEighthRule()}

    if 'method' in var:
        butcher[var['method']]

    elif 'ButcherTableau' in var:
        self.ButcherTableau = var['ButcherTableau']

    else:
        raise ValueError("Invalid Butcher Tableau")
```

Code Listing 5: SetButcherTableau method

This method sets the Butcher Tableau according to the method chosen. The Euler method, Runge-Kutta 2nd order, Runge-Kutta 4th order, and Three-Eighths Rule are implemented. The user can also set the Butcher Tableau manually, by passing the desired rule.

The ConstantK method is shown in Code 6

```python
def ConstantK(self, index, t, y, k)->float:
    a = t + self.c[index] * self.Dt
    b = y + self.Dt * sum([self.a[index][j] * k[j] for j in range(index)])

    return self.ft(a, b)
```

Code Listing 6: ConstantK method

This method is responsible for evaluating the intermediate values of the Runge-Kutta method. The index is the stage of the method, t is the current step time, y is the current value of the function, and k is the list of intermediate values. The return is the function evaluated at the intermediate point.

Finally, the Run method is shown in Code 17

```
1  def Run(self)->None:
2      t = self.t0
3      y = self.y0
4  
5      self.c, self.a, self.b = self.ButcherTableau
6  
7      k = []
8      for _ in range(self.step):
9          for i in range(len(self.c)):
10             k.append(self.ConstantK(i, t, y, k))
11 
12         y += self.Dt * sum([self.b[j] * k[j] for j in range(len(k))])
13         t += self.Dt
14 
15         k.clear()
16         self.sol.append((t, y))
```

Code Listing 7: Run method

The Run method performs the Runge-Kutta method for a given Butcher Tableau. On lines 2 and 3, the initial values of t and y are set. The Butcher Tableau coefficients (c, a, and b) are unpacked on line 5.

From line 8 on, the method iterates over the number of steps. For each step, the intermediate values are calculated on line 10. The new value of y is calculated on line 12, and the new value of t is calculated on line 13. The intermediate values are cleared on line 15, and the t and y values are appended to the sol list on line 16.

2.2 The Galerkin Class

The Galerkin class' attributes are displayed in Code 8.

```python
@dataclass
class Galerkin:
    u_exact: callable
    source_term: callable
    dudx: callable
    dudy: callable
    p_order: int

    n_points: float = field(init=False)
    xi: list[float] = field(init=False, default_factory=list)
    phi: list[float] = field(init=False, default_factory=list)
    dphi: list[float] = field(init=False, default_factory=list)

    points: list[float] = field(init=False, default_factory=list)
    weights: list[float] = field(init=False, default_factory=list)
    pointsBC: list[float] = field(init=False, default_factory=list)
    weightsBC: list[float] = field(init=False, default_factory=list)

    K: list[float] = field(init=False, default_factory=list)
    F: list[float] = field(init=False, default_factory=list)
    alpha: list[float] = field(init=False, default_factory=list)

    error: float = field(init=False, default=0.0)
```

Code Listing 8: Attributes of the Galerkin class

In which u_exact is the exact solution, source_term is the source term, dudx and dudy are the derivatives of the exact solution, and p_order is the polynomial order. The n_points attribute is the number of points used to evaluate the integral. The xi, phi, and dphi lists are the points, basis functions, and derivatives of the basis functions, respectively.

The points and weights are the Gauss-Legendre points and weights, and the pointsBC and weightsBC are the points and weights for the boundary conditions. The K and F lists are the matrices used to solve the linear system of equations, and the alpha list is the solution for the coefficients. The error attribute is the error of the approximation.

The first method is the Run function, which coordinates the Galerkin Method through the other methods. The Run method is shown in Code 9.

```
1 def Run(self)->None:
2     self.Contribute()
3     self.BodyForce()
4     self.ContributeBC()
5 
6     self.alpha = np.linalg.solve(self.K, self.F)
7 
8     self.Error()
```

Code Listing 9: Run method

The Run method is simple. On line 2, it calls the Contribute method, to evaluate the contribution of each integration point to the stiffness matrix. On line 3, the BodyForce method is called so that the contribution of the source term is added to the load vector. Line 4 calls the ContributeBC method, which evaluates the contribution of the boundary conditions. The coefficient alphas are calculated on line 6 as the solution of the linear system. Finally, the error is calculated on line 8.

The Contribute method is shown in Code 10.

```
def Contribute(self)->None:
    self.SetNppoints((self.p_order * 2 + 1)/2)
    self.IntegrationRuleDomain()

    for (x, y), w in zip(self.points, self.weights):
        self.BasisFuntcion(x, y)

        if not len(self.K):
```

```
 9                self.K = np.zeros((len(self.phi), len(self.phi)))
10
11            for i, dphi_i in enumerate(self.dphi):
12                for j, dphi_j in enumerate(self.dphi):
13                    self.K[i, j] += np.dot(dphi_i, dphi_j) * w
```

Code Listing 10: Contribute method

On lines 2 and 3 the integration rule for two dimensions is set. All integration points and respective weights are obtained from the IntegrationRuleDomain method. On line 6, the basis functions are calculated for each integration point. If the stiffness matrix is empty, it is initialized on line 8. The stiffness matrix is calculated on lines 11 to 13.

The code for BasisFunction and IntegrationRuleDomain is not shown here due to its length. However, in List 2 the numerical integration rule was extensively discussed, and the basis functions are calculated from the Legendre polynomials. For more information, see the code in the Appendix A.

The BodyForce method is shown in Code 11.

```
 1 def BodyForce(self)->None:
 2     self.SetNppoints(10)
 3     self.IntegrationRuleDomain()
 4     for (x, y), w in zip(self.points, self.weights):
 5         self.BasisFuntcion(x, y)
 6
 7         if not len(self.F):
 8             self.F = np.zeros(len(self.phi))
 9
10         for i, phi in enumerate(self.phi):
11             self.F[i] += phi * self.source_term(x, y) * w
```

Code Listing 11: BodyForce method

On lines 2 and 3 the integration rule for two dimensions is set. Attention to the fact that, since the source term is a trigonometric function, more integration points are required to evaluate the integral. The basis functions are calculated for each integration point on line 5. If the load vector is empty, it is initialized on line 8. The load vector is calculated on line 11.

The ContributeBC method is shown in Code 12.

```python
def ContributeBC(self)->list[float]:
    self.IntegrationRuleBC()
    for point, w in zip(self.pointsBC, self.BCweights):
        self.BasisFuntcion(1, point)
        for i, phi in enumerate(self.phi):
            self.F[i] += phi * self.dudx(1, point) * w

        self.BasisFuntcion(point, 1)
        for i, phi in enumerate(self.phi):
            self.F[i] += phi * self.dudy(point, 1) * w
```

Code Listing 12: ContributeBC method

On line 2, the integration rule for one dimension is set. The basis functions are calculated for each integration point on line 4. On lines 5 and 6, the boundary condition on the right is imposed and on lines 9 and 10 it is imposed on the top of the domain.

Finally, the Error method calculates the error for the Galerkin Method. The Error method is shown in Code 13.

```
1  def Error(self)->None:
2      self.SetNppoints(10)
3      self.IntegrationRuleDomain()
4  
5      for (x, y), w in zip(self.points, self.weights):
6          self.BasisFuntcion(x, y)
7  
8          u_h = self.alpha @ self.phi
9  
10         self.error += ((self.u_exact(x, y) - u_h) ** 2) * w
11 
12     self.error = np.sqrt(self.error)
```

Code Listing 13: Error method

2.3 The LeastSquare Class

The LeastSquare class' attributes are displayed in Code 14.

```python
@dataclass
class LeastSquare:
    x: list
    y: list
    approximation_type: str

    order: int = 1

    K: list = field(init=False, default_factory=list)
    F: list = field(init=False, default_factory=list)
    alpha: list = field(init=False, default_factory=list)

    approx_solution: list = field(init=False, default_factory=list)

    errors: list = field(init=False, default_factory=list)
    total_error: float = field(init=False, default=0.0)
```

Code Listing 14: Attributes of the LeastSquare class

Where x and y are the set of data points, the approximation type defines whether the approximation will be polynomial, logarithmic or non-linear. If a polynomial is set, an order can be chosen as well. The K and F are the matrices used to solve the linear system of equations and alpha is the solution for the coefficients. The approx_solution is the function evaluated at the data points, reconstructed by using the alpha coefficients, the errors list is the error for each data point, and the total_error is the sum of all errors.

The class contains two set methods: SetOrder, for the polynomial approximation order, and SetMethod, for the approximation type. There are also the PolynomialApproximation, LogarithmicApproximation, and NonLinearApproximation methods, which are used to find the alpha coefficients for each approximation type.

The PolynomialApproximation method is shown in Code 15.

```python
def PolynomialApproximation(self)->None:
    n = self.order + 1
    m = len(self.x)

    self.K = np.zeros((n, n))
    self.F = np.zeros(n)

    for i in range(n):
        self.F[i] = sum([self.y[k] * self.x[k] ** i for k in range(m)])

        for j in range(n):
            self.K[i, j] = sum([self.x[k] ** (i + j) for k in range(m)])
```

Code Listing 15: PolynomialApproximation method

The LogarithmicApproximation method is shown in Code 16.

```python
def LogarithmicApproximation(self)->None:
    self.y = np.log(self.y)
    self.x = np.log(self.x)

    self.PolynomialApproximation()

    self.y = np.exp(self.y)
    self.x = np.exp(self.x)
```

Code Listing 16: LogarithmicApproximation method

Note that, since the logarithmic approximation linearizes the data, the y and x values are transformed to their natural logarithm. After the approximation is found, the y and x values are transformed back to their original values. The NonLinearApproximation method is similar to the NonLinearSolver class, presented in List 5. For more information, see the code in the Appendix A.

The Run method called in the main function is the one responsible for maintaining the flow of the program. It is shown in Code 17.

```python
def Run(self)->None:
    method = {
```

```
3          "Polynomial": self.PolynomialApproximation,
4          "Logarithmic": self.LogarithmicApproximation,
5          "NonLinear": self.NonLinearApproximation
6      }
7
8      method[self.approximation_type]()
9
10     if self.approximation_type in ["Polynomial", "Logarithmic"]:
11         self.Solver()
12
13     self.CalcApproxSolution()
14
15     self.Error()
```

Code Listing 17: Run method

On lines 2 to 6, a dictionary is created to relate the approximation type to the method that approximates the data points. On line 8, the method is called. If the approximation type is polynomial or logarithmic, the Solver method is called on line 11. The approximated solution is evaluated on line 13 and the error is calculated on line 15.

The Solver method is presented in Code 18

```
def Solver(self)->None:
    self.alpha = np.linalg.solve(self.K, self.F)

    if self.approximation_type == "Logarithmic":
        self.alpha[0] = np.exp(self.alpha[0])
```

Code Listing 18: Solver method

Note that, if the approximation is logarithmic, the alpha coefficient "b" is transformed to its exponential value. The CalcApproxSolution method is shown in Code 19.

```
def CalcApproxSolution(self)->None:
    m = len(self.x)
    n = self.order + 1
```

```python

    self.approx_solution = np.zeros(m)

    for i in range(m):
        if self.approximation_type == "Polynomial":
            self.approx_solution[i] = sum([self.alpha[j] * self.x[i] ** j for j in range(n)])

        elif self.approximation_type == "Logarithmic":
            self.approx_solution[i] = self.alpha[0] * self.x[i] ** self.alpha[1]

        elif self.approximation_type == "NonLinear":
            self.approx_solution[i] = self.alpha[1] * self.x[i] ** self.alpha[0]

        else:
            raise ValueError("Invalid approximation type")
```

Code Listing 19: CalcApproxSolution method

The CalcApproxSolution method evaluates the approximated solution for each data point depending on the approximation type. Finally, the Error method calculates the error for each data point and the total error. The Error method is shown in Code 20.

```python
def Error(self)->None:
    m = len(self.x)

    self.errors = np.zeros(m)

    for i in range(m):
        self.errors[i] = (self.y[i] - self.approx_solution[i]) ** 2

    self.total_error = sum(self.errors)
```

Code Listing 20: Error method

3 Results

The results for exercises 1, 2, and 3 are briefly presented in the following sections.

3.1 Exercise 1

For exercise 1, the Runge-Kutta method was implemented using the Butcher Tableau. As a result, the approximated value for the ordinary differential equation is obtained for each time step. The results are displayed in Table 3.1.

Table 3.1: Results for the Runge-Kutta method

x	y
2.0	1.0
2.1	1.19091
2.2	1.36667
2.3	1.53077
2.4	1.68571
2.5	1.83333
2.6	1.97500
2.7	2.11176
2.8	2.24444
2.9	2.37368
3.0	2.50000

3.2 Exercise 2

For exercise 2, Table 3.2 presents the α_i coefficients for the approximation of u_h depending on the polynomial order n.

Table 3.3 shows the error for the approximation of u_h for each polynomial order n, which is also depicted in Figure 3.1.

Table 3.2: Results for the approximation of u_h

$\boldsymbol{\alpha}$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
α_1	-4.33e-9	0.6320	0.4918	0.4927	0.4981
α_2	-	-0.6108	-0.3698	-0.3713	-0.3814
α_3	-	-0.6108	-0.1215	-0.1198	-0.1077
α_4	-	0.5685	-0.3698	-1.47e-3	-1.90e-2
α_5	-	-	0.2780	-0.3713	1.00e-2
α_6	-	-	0.0904	0.2795	-0.3814
α_7	-	-	-0.1215	9.00e-2	0.2922
α_8	-	-	0.0904	1.45e-3	8.24e-2
α_9	-	-	0.03232	-0.1198	1.47e-2
α_{10}	-	-	-	9.00e-2	-7.70e-3
α_{11}	-	-	-	2.99e-2	-0.107
α_{12}	-	-	-	-1.65e-4	8.24e-2
α_{13}	-	-	-	-1.47e-3	2.31e-2
α_{14}	-	-	-	1.45e-3	4.00e-3
α_{15}	-	-	-	-1.65e-4	-2.14e-3
α_{16}	-	-	-	1.37e-3	-1.90e-2
α_{17}	-	-	-	-	1.47e-2
α_{18}	-	-	-	-	4.00e-3
α_{19}	-	-	-	-	9.26e-4
α_{20}	-	-	-	-	-4.22e-4
α_{21}	-	-	-	-	1.00e-2
α_{22}	-	-	-	-	-7.70e-3
α_{23}	-	-	-	-	-2.14e-3
α_{24}	-	-	-	-	-4.22e-4
α_{25}	-	-	-	-	4.01e-4

3.3 Exercise 3

As the main results for exercise 3, the coefficients for the curve fitting (Eq. (1.17)) are presented in Table 3.4, as well as the total error for the approximation.

Table 3.3: Error for the approximation of u_h vs. polynomial order n

n	Error
1	0.673542425905569
2	0.11149814453502095
3	0.006645639492674947
4	0.006412141835255881
5	0.0021858066887519693

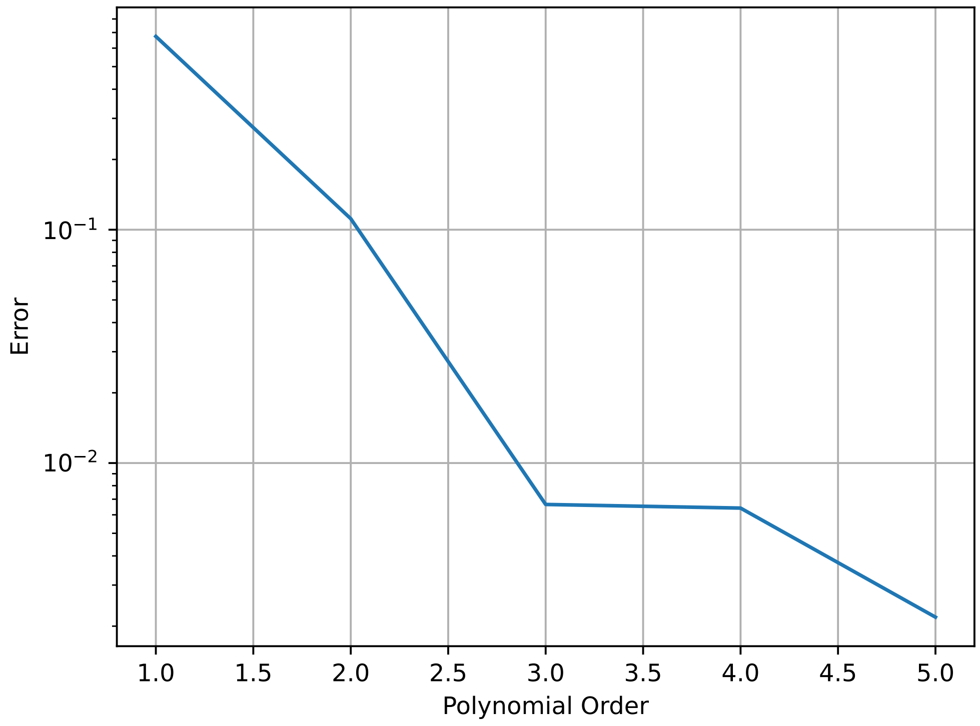

Figure 3.1: Error for the approximation of u_h vs. polynomial order n

Table 3.4: Results for the curve fitting

Approximation Type	**Coefficient a**	**Coefficient b**	**Curve fitting**	**Total Error**
Logarithmic	0.5974	1.2821	$1.2821x^{0.5974}$	24.2828
Non-linear	0.5974	1.1804	$1.1804x^{0.5974}$	19.7567

As expected, the linearization of the logarithmic function provides a worse approximation than the non-linear method, since the latter is the solution of the Least Squares Method. Figure 3.2 shows the data points and the approximated solution for both approximation types.

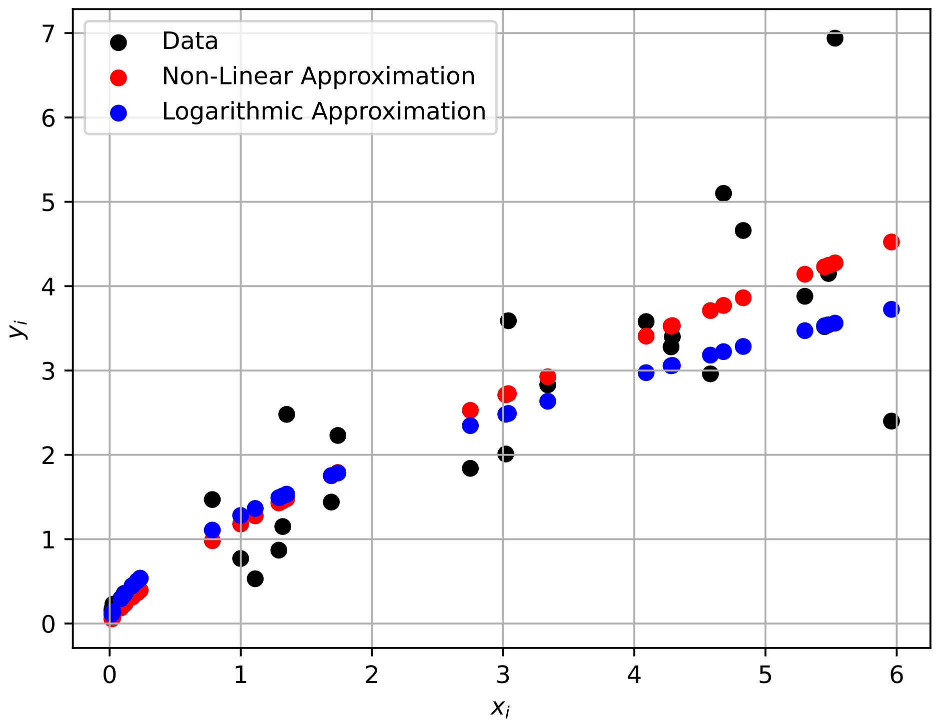

Figure 3.2: Curve fitting for the data points using the Least Squares Method

4 Conclusions

The present work aimed to solve the ordinary differential equation using the Runge-Kutta method. An algorithm is developed to read a given Butcher Tableau and approximate the solution for the ODE. The main difference between initial value problems and boundary value problems is discussed, and for the latter, a Galerkin algorithm is also implemented.

For the Galerkin method, a few comments are in order. Since the method is based on the integration of functions, the number of integration points when using the Gaussian Quadrature is key to the accuracy of the results. During the elaboration of the code, it was observed that, for a trigonometric body force and exact solution, the method suffered considerably from integrating accurately the functions. Only when 10 integration points were set the results were close to the exact solution.

Another important aspect is that the domain herein used is coincident with the domain in which the Gaussian Quadrature is defined. Otherwise, the method would require the evaluation of the Jacobian to correctly integrate the functions changing the domain of integration. The Boundary Conditions imposed in the problem are also crucial, being the Dirichlet condition null on $\partial\Omega_D$ so that no extra BC methods are implemented. The Neumann BC is simply the inner product of the gradient of the state variable u with the normal vector $\boldsymbol{n}$ to the edge of the domain.

Finally, it is observed that the Least Squares Method indeed yields a more accurate solution than the linearization of the curve fitting the set of data points. Although the method for non-linear systems is more computationally expensive, its employment is justified when the linearization of the curve is not accurate enough.

References

1 CUNHA, M. C. de C. *Métodos numéricos*. [S.l.]: Editora da UNICAMP, 2000.

2 BURDEN, R. L.; FAIRES, J. D. *Numerical analysis, brooks*. [S.l.]: Cole publishing company Pacific Grove, CA:, 1997.

A GitHub Repository

The source code for this report and every code mentioned can be found in the following GitHub repository: CarlosPuga14/MetodosNumericos_2024S1.