UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Civil, Arquitetura e Urbanismo

IC639: Métodos Numéricos para Engenharia Civil

List 2

Numerical Integration

Student:

Carlos Henrique Chama Puga - 195416

Advisors:

Porf. Dr. Philippe Devloo

Dr. Giovane Avancini

Campinas

2024

Contents

1 Introduction

The need of numerically integrate a function arises for evaluate the definite integral of a function that might have explicit antiderivative or whose antiderivatiev is not easy to obtain. In numerical methods, such as the Finite Element Method, the numerical integration is used to evaluate the integration that composes the weak formulation of the problem.

The main ideia behind numerical integration is to approximate the integral of a function by the sum of weights times the function evaluated at some points (see Eq. (1.1))

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i). \tag{1.1}$$

in which w_i are the weights and x_i are the points where the function is evaluated.

This list approaches several techiniques designed to numerically integrate a function. Herein are comprehended the Trapzoidal, Simpson's (1/3 and 3/8) and Gauss-Legendre quadrature rules. Equations (1.2), (1.3), (1.4) and (1.5) show the formulas for each method, respectively.

$$\int_a^b f(x)dx \approx b-a\left[\frac{1}{2}f(a)+\frac{1}{2}f(b)\right], \tag{1.2}$$

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\left[\frac{1}{3}f(a)+\frac{4}{3}f\left(\frac{a+b}{2}\right)+\frac{1}{3}f(b)\right], \tag{1.3}$$

$$\int_a^b f(x)dx \approx (b-a)\left[\frac{1}{8}f(a)+\frac{3}{8}f\left(\frac{2a+b}{3}\right)+\frac{3}{8}f\left(\frac{a+2b}{3}\right)+\frac{1}{8}f(b)\right], \tag{1.4}$$

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \text{DetJac} f\left(x(\xi_i)\right) w_i, \tag{1.5}$$

in which, for Eq. (1.5), the mapping function $x(\xi_i)$ is given by Eq. (1.6)

$$x(\xi_i) = \frac{1-\xi_i}{2}a + \frac{1+\xi_i}{2}b, \tag{1.6}$$

and DetJac is the Jacobian determinant, given by Eq. (1.7)

$$\text{DetJac} = \frac{b-a}{2}, \tag{1.7}$$

The following refereces were used during this report elaboration: (1), (2), and (3).

2 Problem Statement

For this work, three different functions are chosen to be numerically integrated. These functions are presented in this section, along with their analytical solutions. To integrate the functions the Trapezoidal, the Simpson's 1/3 and 3/8, and the Gauss-Legendre rules are employed. The results obtained by these methods are compared with the analytical solutions, and the error convergence is analyzed.

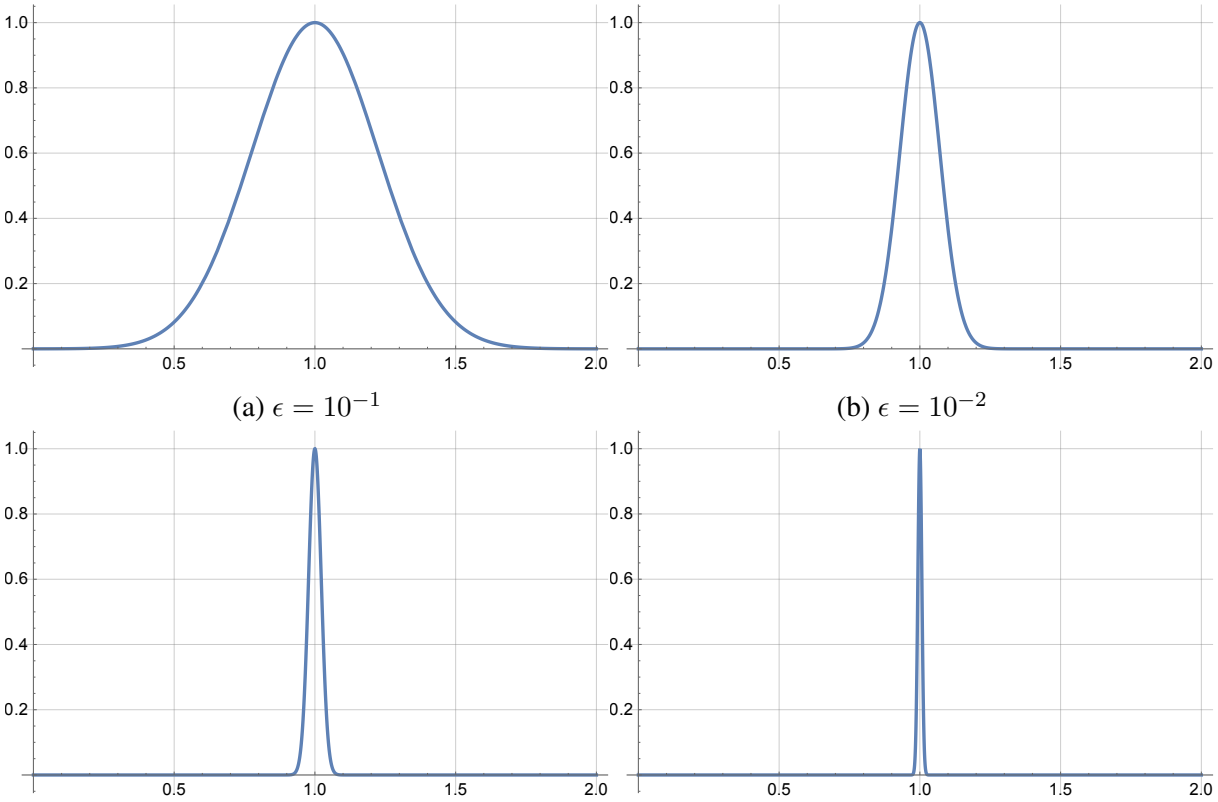
For every function, one presents the function itself and the analytical solution for the integral. A code in Mathematica is developed to help with the visualization.

2.1 Function 1

Function 1 is represented by Eq. (2.1)

$$f(x) = e^{-\frac{(x-1)^2}{\epsilon}} \quad \forall x \in \mathbb{R} | 0 \leq x \leq 2, \epsilon \to 0, \tag{2.1}$$

since it depends on the parameter ϵ, a simple analysis over its behaviour its made. Fig. 2.1 shows how it is the variation of the function for differente values of ϵ.



(a) $\epsilon = 10^{-1}$ (b) $\epsilon = 10^{-2}$

(c) $\epsilon = 10^{-3}$ (d) $\epsilon = 10^{-4}$

Figure 2.1: Function 1

By analyzing Fig. 2.1, one can note that, the lesser the value of ϵ, the more the function behaves like a Dirac Delta function, which is not integrable. The analytical solution for the integral of the function is given by Eq. (2.2)

$$\int e^{-\frac{(x-1)^2}{\epsilon}} \, dx = \frac{\sqrt{\pi}\, \text{Erf}\left(c(-1+x)\right)}{2c}, \tag{2.2}$$

in which c is a constant given by $c = pot\sqrt{10}$, and pot is a integer number found in $\epsilon = 10^{-pot}$. Erf is the error function, defined by Eq. (2.3)

$$\text{Erf}(z) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{z}{2n+1} \prod_{k=0}^{n} \frac{-z^2}{k}. \tag{2.3}$$

In this work, the error function is evaluated using the Python library `scipy`.

2.2 Function 2

Function 2 is given by Eq. (2.4) and Fig. 2.2 shows its behaviour

$$f(x) = x \sin\left(\frac{1}{x}\right) \quad \forall x \in \mathbb{R} | 1/100 \leq x \leq 1/10. \tag{2.4}$$

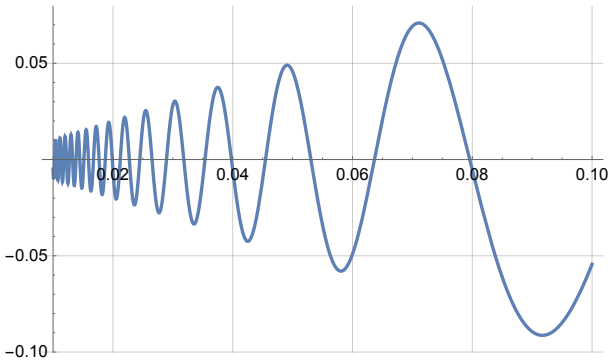


Figure 2.2: Function 2

For this function, the analytical solution for the integral is given by Eq. (2.5)

$$\int x \sin\left(\frac{1}{x}\right) \, dx = \frac{x}{2} \cos\left(\frac{1}{x}\right) + \frac{x^2}{2} \sin\left(\frac{1}{x}\right) + \frac{1}{2}\text{Si}\left(\frac{1}{x}\right), \tag{2.5}$$

in which Si is the sine integral function, defined by Eq. (2.6)

$$\text{Si}(z) = \int_0^z \frac{\sin(t)}{t} \, dt. \tag{2.6}$$

Eq. (2.6) can be approximated by the Padé approximants of the convergent Taylor Series, available in here.

2.3 Function 3

Finally, Function 3 is given by Eq. (2.7) and Fig. 2.3 shows its behaviour

$$f(x) = \begin{cases} 2x + 5 & 0 \leq x \leq 1/\pi \\ \frac{-5\pi^2(x^2-2x-5)+10\pi x+2x}{1+5\pi^2} & 1/\pi \leq x \leq 2/\pi \cdot \\ 2\sin(2x) + \frac{4+20\pi^2+25\pi^3}{\pi+5\pi^3} - 2\sin\left(\frac{4}{\pi}\right) & 2/\pi \leq x \leq 8/\pi \end{cases} \tag{2.7}$$

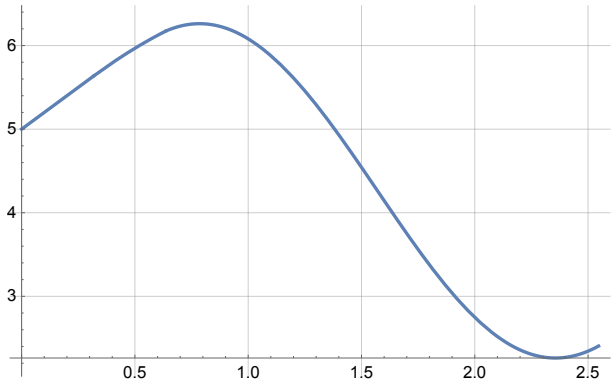


Figure 2.3: Function 3

Since the antiderivative of this function is a piecewise function, the analytical solution for the integral is given by parts as well (see Eq. (2.8))

$$\int f(x)\,dx = \begin{cases} x^2 + 5x & 0 \leq x \leq 1/\pi \\ \frac{25\pi^2 x+x^2+5\pi x^2+5\pi^2 x^2-\frac{5\pi^2 x^3}{3}}{1+5\pi^2} & 1/\pi \leq x \leq 2/\pi \cdot \\ \frac{(4+20\pi^2+25\pi^3)x}{\pi+5\pi^3} - \cos(2x) - 2x\sin\left(\frac{4}{\pi}\right) & 2/\pi \leq x \leq 8/\pi \end{cases} \tag{2.8}$$

3 Code Implementation

In this section, a brief explanation of how the numerical methods for integration are implemented. The code is written in Python, and a total of six classes are developed, besides the main function file. Although parts of the code are presented here, the complete code is available on the GitHub repository, which can be found in the Appendix A.

3.1 Main File

The main function is written in the `main.py` file. This file is responsible for structuring the code and creating objects for numerical integration analysis. An example of implementation for the main function is shown in Code 1.

```
def main():
    n_divisions = 2
    npoints = 4
    interval = Interval(0.0, 1.0, n_divisions, npoints)

    # defining the refinement level
    ref_level = 2

    # defining the function and its exact solution
    p = 1
    func = lambda x: x ** p
    exact = lambda x: 1/(p + 1) * x ** (p + 1)

    # setting the integration rule
    interval.SetGaussLegendreRule()

    # integrating the function
    interval.NumericalIntegrate(func, ref_level)

    # computing the error
    interval.SetExactSolution(exact, ref_level)
    interval.ComputeError()
```

```
23
24      return 0
```

Code Listing 1: Main function.

In line 1 the Interval class is imported, and in line 7 an object of this class is created. As input data, it takes the limits of the interval (in this case 0 and 1), the number of divisions, and the number of integration points.

The function to be integrated is then defined, here is shown the polynomial x, and its exact solution, $0.5x^2$. The integration rule is set in line 18, and the function is integrated in line 21. In this example, it is used the Gauss-Legendre integration rule.

The exact solution is set in line 24, and the error is computed in line 25. The results can also be printed in a `.txt` file, which is not presented here.

3.2 The Interval Class

The Interval class was developed by Giovane Avancini. However, a few modifictations were made to adapt it to the project's structure. This is the main class of this work, responsible for dealing with the interval data structure and computing the numerical integration.

The fields from the Interval class are shown in Code 2. Note that, since the dataclass decorator is used, the fields are automatically initialized in the constructor, not being necessary to define it explicitly.

```
1 from dataclasses import dataclass
2
3 @dataclass
4 class Interval:
5     total_error: ClassVar[float] = 0.0
6     _a: float
7     _b: float
8     _n_refinements: int
```

```
    _refinement_level: int = field(default=0)
    _n_points: int = field(default=0)
    _method: IntegrationRule = field(init=False, default=None)
    _sub_intervals: list = field(init=False, default_factory=list)
    _numerical_integral: float = field(init=False, default=0.0)
    _analytic_integral: float = field(init=False, default=0.0)
    _integration_error: float = field(init=False, default=0.0)
```

Code Listing 2: Interval class fields.

The floats a and b represents the integration interval, $n\_refinement$ is the number of refinements that is going to be made, and $refinement\_level$ is the current interval refinement level. The n_points variable is the number of points used in the numerical integration (mostly used in the Gauss-Legendre rule). A class variable, $total\_error$, is used to store the integration total error.

From the $method$ field on, the fields are not initialized at the moment of the object creation, which explains the `init=False` argument. The $method$ field is an object from the IntegrationRule class, $sub\_intervals$ is a list that might contain the interval subdivisions, $numerical\_integral$ is the result of the numerical integration, $analytic\_integral$ is the exact solution, and the error between the numerical and the exact solution is stored in $integration\_error$.

Among the methods of the Interval class, one stands out: the NumericalIntegrate method. This method is responsible for computing the numerical integration of a given function. The code for this method is shown in Code 3.

```
def NumericalIntegrate(self, func: callable, ref_level: int = 0)->float:
    self.numerical_integral = 0.0
    if self.refinement_level == ref_level:
        self.numerical_integral = self.method.Integrate(func, self.a, self.
    b, self.n_points)
    else:
        for interval in self.sub_intervals:
            self.numerical_integral += interval.NumericalIntegrate(func,
    ref_level)
```

```
8
9       return self.numerical_integral
```

Code Listing 3: NumericalIntegrate Method.

Line 3 checks if the current interval is the one that the numerical integration is going to be computed. If it is, the method Integrate from the IntegrationRule class is called. Otherwise, the method is called recursively for each subinterval.

Finally, the ComputeError method (Code 4) is responsible for computing the error between the numerical and the exact solution.

```
1  def ComputeError(self)->None:
2      if ref_level == 0:
3          self.integration_error = abs(self.analytic_integral - self.
       numerical_integral)
4          Interval.total_error += self.integration_error
5
6      interval: Interval
7      for interval in self.sub_intervals:
8          if interval.refinement_level == ref_level:
9              interval.integration_error = abs(self.analytic_integral - self.
       numerical_integral)
10             Interval.total_error += interval.integration_error
11
12         interval.ComputeError(ref_level)
```

Code Listing 4: ComputeError Method

Note that there is a difference between the variable $interval$ and the class $Interval$. To be able to access the variable class $total\_error$, the class name is used. One cite the other methods from the Interval class, such as the Setters and Getetrs, responsible for setting and getting the class fields; the set of functions used to set the integration rule, and the method used to set the exact solution, which has been slightly modified from the original code, in order to accept a refinement level as an argument.

3.3 The IntegrationRule Class

The IntegrationRule class is what it's called an *abstract class*. It is a class that cannot be instantiated, but it is used as a base class for other classes. In this case, it is used as a base class for the numerical integration rule classes.

As its abstract methods, one can cite the IntegrationPoints method, in which the integration points are computed, the Xmap method, which maps the integration points to the interval, and the DetJac method, which computes the Jacobian determinant. An observation is made at this point. The methods' names might not be appropriate for the one-third and three-eighths Simpson, and the Trapezoidal rules. However, the names come from the ones employed in Gauss-Legendre implementations.

This class takes in advantage the fact that, for all numerical methods herein presented, the integration process (see Eq. (1.1)) is the same, only the integration points and weights change. Using this, the integration process is implemented in the Integrate method, shown in Code 5.

```
1 def Integrate(self, func: callable, a: float, b: float, n_points: int)->
     float:
2     self.IntegrationPoints(a, b, n_points)
3 
4     integral = 0.0
5     for point, weight in zip(self.points, self.weights):
6         self.ComputeRequiredData(point, a, b)
7 
8         integral += self.detJac * func(self.Xmapped) * weight
9     return integral
```

Code Listing 5: Integrate Method.

The function ComputeRequiredData, in line 6, is responsible for computing the mapping of the integration points to the interval, and the Jacobian determinant. Again, it is emphasized the fact that the Simpsons and Trapezoidal rules

do not have a Jacobian determinant, but the method name is kept for consistency. Instead, it is used the constant that multiplies the sum of the function evaluations.

3.4 The SimpsonOneThirdRule Class

The SimpsonOneThirdRule class (Code 6) is where the method is properly developed. It inherits from the IntegrationRule class, and the methods IntegrationPoints, Xmap, and DetJac are implemented.

```
1  from IntegrationRule import IntegrationRule
2
3  @dataclass
4  class SimpsonOneThirdRule(IntegrationRule):
5      def IntegrationPoints(self, a: float, b: float, _)->None:
6          self.points = [a, (a+b)/2, b]
7          self.weights = [1/3, 4/3, 1/3]
8
9      def Xmap(self, xi: float, _, __)->None:
10         self.Xmapped = xi
11
12     def DetJac(self, a: float, b: float)->float:
13         self.detjac = (b - a) / 2
```

Code Listing 6: SimpsonOneThirdRule Class.

In line 5, the third argument passed to the function is not used, the reason why it is written as an underscore. Note that by using the points and weights as implemented, the final equation for the Simpson's 1/3 rule is in concordance with Eq. (1.3).

In this rule, there is no mapping function, that is why the Xmap returns the same value as the input. For the same reason, the DetJac returns the multiplication constant for the rule.

3.5 The SimpsonThreeEighthsRule Class

Simarly to the SimpsonOneThirdRule class, the SimpsonThreeEighthsRule class (Code 7) is implemented.

```
from IntegrationRule import IntegrationRule

@dataclass
class SimpsonThreeEighthsRule(IntegrationRule):
    def IntegrationPoints(self, a: float, b: float, _) -> None:
        self.points = [a, (2* a + b) / 3, (a + 2 * b) / 3, b]
        self.weights = [1/8, 3/8, 3/8, 1/8]

    def Xmap(self, xi: float, _, __) -> None:
        self.Xmapped = xi

    def DetJac(self, a: float, b: float)-> float:
        self.detjac = (b - a)
```

Code Listing 7: SimpsonThreeEighthsRule Class.

3.6 The TrapezoidalRule Class

The TrapezoidalRule class (Code 8) is implemented in the same way as the previous classes.

```
from IntegrationRule import IntegrationRule

@dataclass
class TrapezoidalRule(IntegrationRule):
    def IntegrationPoints(self, a: float, b: float, _)->None:
        self.points = [a, b]
        self.weights = [1/2, 1/2]

    def Xmap(self, xi: float, _, __)->None:
        self.Xmapped = xi

    def DetJac(self, a: float, b: float)->float:
```

```
13          self.detjac = (b - a)
```

Code Listing 8: TrapezoidalRule Class.

3.7 The GaussLegendreRule Class

The last class implemented in this work. The main difference between the GaussLegendreRule (Code 9) and the others is the existence of a mapping function and a Jacobian determinant.

The number of points is a parameter for the IntegrationPoints method, which defines the points and weights to be used during the integration.

```
1  from IntegrationRule import IntegrationRule
2
3  @dataclass
4  class GaussLegendreRule(IntegrationRule):
5      def IntegrationPoints(self, _, __, npoints: int) -> None:
6          match npoints:
7              ...
8              case 4:
9                  self.weights.append(0.2369268850561891)
10                 self.weights.append(0.4786286704993665)
11                 self.weights.append(0.5688888888888889)
12                 self.weights.append(0.4786286704993665)
13                 self.weights.append(0.2369268850561891)
14
15                 self.points.append(-0.9061798459386640)
16                 self.points.append(-0.5384693101056831)
17                 self.points.append(0)
18                 self.points.append(0.5384693101056831)
19                 self.points.append(0.9061798459386640)
20
21             case _:
22                 raise ValueError("Invalid number of points")
23
24     def Xmap(self, xi: float, a: float, b: float) -> None:
25         self.Xmapped = a * (1 - xi) / 2 + b * (1 + xi) / 2
26
```

```
27    def DetJac(self, a: float, b: float) -> float:
28        self.detjac = (b - a) / 2
```

Code Listing 9: GaussLegendreRule Class.

Herein is shown the implementation for the four-point Gauss-Legendre rule, but the three dots on the top indicates the points and weights for 1, 2, and 3 points rules. The weights and points are defined in lines 9 to 19. The mapping function is implemented in line 24, and the Jacobian determinant in line 27.

With these classes implemented, the numerical integrations can be executed. The following section presents the results obtained during this work.

4 Results

This section presents the main results achieved by the proposed methods. The results are divided into two parts: the first one presents the values obtained by the numerical integration using the methods implemented in this work, using 4 refinement levels. The second part compares the numerical and the analytical results, discussing whether or not the error convergence is as expected.

4.1 Numerical Integration Results

4.2 Comparison between Numerical and Analytical Results

5 Conclusion

References

1 BECKER, E.; CAREY, G.; ODEN, J. Finite elements: An introduction. Prentice-Hall, n. v. 1, 1981.

2 BURDEN, R. L.; FAIRES, J. D. Student solutions manual and study guide: numerical analysis. *(No Title)*, 2005.

3 CUNHA, M. *Métodos Numéricos*. [S.l.]: UNICAMP. ISBN 9788526808775.

A GitHub Repository

The source code for this report and every code inhere mentioned can be found in the following GitHub repository: CarlosPuga14/MetodosNumericos_-2024S1.