UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Civil, Arquitetura e Urbanismo

IC639: Métodos Numéricos para Engenharia Civil

# List 2

# Numerical Integration

**Aluno:**

Carlos Henrique Chama Puga - 195416

**Docentes**:

Porf. Dr. Philippe Devloo

Dr. Giovane Avancini

Campinas

2024

# Contents

# 1 Introduction

# 2 Problem Presentation

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 these functions, the following methods are employed: the Trapezoidal Rule, the Simpson's 1/3 Rule, the Simpson's 3/8 Rule, and the Gauss-Legendre Quadrature. The results obtained by these methods are compared with the analytical solutions, and the error convergence is analyzed.

## 2.1 Function 1

## 2.2 Function 2

## 2.3 Function 3

# 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 file containing the main function. Although parts of the code are presented here, the complete code is available on the GitHub repository, available 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, creating objects for numerical integration analysis. An example of implementation for the main file is shown in Code 1.

```
1  def main():
2      n_divisions = 2
3      npoints = 4
4      interval = Interval(0.0, 1.0, n_divisions, npoints)
5
6      # defining the refinement level
7      ref_level = 2
8
9      # defining the function and its exact solution
10     p = 1
11     func = lambda x: x ** p
12     exact = lambda x: 1/(p + 1) * x ** (p + 1)
13
14     # setting the integration rule
15     interval.SetGaussLegendreRule()
16
17     # integrating the function
18     interval.NumericalIntegrate(func, ref_level)
19
20     # computing the error
21     interval.SetExactSolution(exact, ref_level)
22     interval.ComputeError()
23
24     return 0
```

Code Listing 1: Main function.

It is emphasized that this is just a simple example. 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.

A 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. The next sections will present, in more detail, the classes devloped in this project.

## 3.2 The Interval Class

The Interval class was developed by the Ph.D. Giovane Avancini. However, a few modifictations were made to adapt it to the project. 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. Notice that, since the dataclass decorator is used, the fields are automatically initialized in the constructor, not being necessary to define it explicitly.

```
from dataclasses import dataclass

@dataclass
class Interval:
    _a: float
    _b: float
    _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 $n\_points$ is the number of points used in the numerical integration (mostly used in the Gauss-Legendre rule).

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.

```
1 def NumericalIntegrate(self, func: callable, ref_level: int = 0)->float:
2     self.numerical_integral = 0.0
3     if self.refinement_level == ref_level:
4         self.numerical_integral = self.method.Integrate(func, self.a, self.
    b, self.n_points)
5     else:
6         for interval in self.sub_intervals:
7             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)->float:
2     self.integration_error = abs(self.numerical_integral - self.
    analytic_integral)
3 
4     interval: Interval
5     for interval in self.sub_intervals:
6         interval.ComputeError()
```

```
    return self.integration_error
```

Code Listing 4: ComputeError Method

## 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 rules.

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' name might not be appropriate for the one-third and three-eighths Simpson, and the Trapezoidal rules. However, the names come from the usually employed in Gauss-Legendre implementations.

This class takes in advantage the fact that, for all numerical methods herein presented, the integration process (see Eq. (**??**)) 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.

```
def Integrate(self, func: callable, a: float, b: float, n_points: int)->
    float:
    self.IntegrationPoints(a, b, n_points)

    integral = 0.0
    for point, weight in zip(self.points, self.weights):
        self.ComputeRequiredData(point, a, b)

        integral += self.detJac * func(self.Xmapped) * weight
    return integral
```

Code Listing 5: Integrate Method.