Calculo de Integrais

Método do Trapézio

Joao Felipe Bianchi Curcio
Jonas Edward Tashiro
Luan Lopes Barbosa de Almeida
Rafael Melloni Chacon Arnone

Contents

1 Organização de Pastas

```
src
├── exp
│   ├── definitions.h
│   ├── expression.h
│   ├── lexicon.h
│   ├── token.h
│   ├── lexicon.cpp
│   ├── expression.cpp
│   └── token.cpp
├── integral
│   ├── integral.h
│   └── integral.cpp
├── Makefile
└── main.cpp
```

2 Conteudo dos Arquivos

2.1 Pasta src

Listing 1: main.cpp

```cpp
#include "exp/expression.h"
#include "exp/token.h"
#include "integral/integral.h"
#include <ios>
#include <ostream>
#include <string>
#include <iostream>
#include <utility>
#include <vector>
#include <limits>
#include <iomanip>

int main() {
    bool quit = false;
    bool cannot_be_conv = false;
    std::string user_expr;
    std::string user_info1, user_info2;
    int numTrapz, precision;
    double start, end;
    std::vector<std::pair<double, double>> fnTable;

    Expression *expression;
    do
    {
        std::cout << "\nEscreva uma expressao:\n";
        std::cout << "f(x) = ";
        std::getline(std::cin, user_expr);
        user_expr += ';';
        std::endl(std::cout);

```

```
31          if(!addSpaces(user_expr))
32          {
33              expression = new Expression(user_expr);
34              if(expression->isValid())
35              {
36                  expression->infixToPostfix();
37                  do{
38                      std::cout << "\nEscreva o ponto extremo da esquerda:\n";
39                      std::cin >> user_info1;
40                      std::cout << "Escreva o ponto extremo da direita:\n";
41                      std::cin >> user_info2;
42                      try{
43                          start = std::stold(user_info1.data());
44                          end = std::stold(user_info2.data());
45                          cannot_be_conv = false;
46                          user_info1.clear(), user_info2.clear();
47                      }catch (std::invalid_argument){
48                          std::cout << "Uma variavel foi escrita de
49                                       forma indevida";
50                          cannot_be_conv = true;
51                      }
52                  }while (cannot_be_conv);
53
54                  do {
55                      std::cout << "\nEscreva o numero de casas decimais\n";
56                      std::cin >> user_info1;
57                      std::cout << "Escreva o numero de trapezios:\n";
58                      std::cin >> user_info2;
59                      try {
60                          precision = std::stoi(user_info1.data());
61                          numTrapz = std::stoi(user_info2.data());
62                          cannot_be_conv = false;
63                          user_info1.clear(), user_info2.clear();
64                      } catch (std::invalid_argument) {
65                          std::cout << "Uma variavel foi escrita de
66                                       forma indevida";
67                          cannot_be_conv = true;
68                      }
69                  }while (cannot_be_conv);
70
71                  std::cout << std::fixed;
72                  std::cout << std::setprecision(precision + 1);
73
74                  TrapezoidIntegral integralCalc(start,end,numTrapz,precision);
75                  integralCalc.calculateIntegral(*expression, fnTable);
76
77                  std::cout << "\n\nSaida:\n\n";
78                  std::cout << "Erro de arredondamento = "
79                            << integralCalc.errorRounding;
80
81                  std::cout << std::fixed;
82                  std::cout << std::setprecision(precision);
83
84                  std::cout << "\nValor da Integral = "
85                            << integralCalc.sumTraps;
86                  std::cout << "\nTabela de Valores:\n";
87                  std::cout << 'x';
88
89                  for (int i = -1; i <= precision; i++)
90                      std::cout << ' ';
```

```
91
92                  std::cout << "| " << "f(x)" << '\n';
93                  for (auto f : fnTable)
94                      std::cout << f.first << " | " << f.second << '\n';
95                  delete expression;
96              }
97              else
98                  std::cout << "\nA expressao fornecida contem erro\n";
99          }
100         else
101             std::cout << "\nErro Lexico detectado na expressao fornecida\n";
102
103         std::cout << "Deseja sair do programa?\n";
104         std::cout << "Sim (Escreva q)\t Nao (Escreva qualquer coisa)\n";
105         std::cin >> user_info1;
106         if(user_info1 == "q")
107             quit = true;
108
109         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),'\n');
110         user_expr.clear();
111         user_info1.clear();
112         std::cout << "\033[2J\033[1;1H";
113
114     }while (!quit);
115     return 0;
116 }
```

Este proximo arquivo tem como intuito facilitar o processo de compilação.

Listing 2: Makefile

```
EXPRESSION = ./exp
INTEGRAL = ./integral
SOURCES = main.cpp
SOURCES += $(EXPRESSION)/expression.cpp $(EXPRESSION)/lexicon.cpp
SOURCES += $(EXPRESSION)/token.cpp
SOURCES += $(INTEGRAL)/integral.cpp
OBJECTS = $(addsuffix .o, $(basename $(notdir $(SOURCES))))


COMPILE : LINK
    g++ -o main $(OBJECTS)

LINK:
    g++ -c $(SOURCES)

clean:
    rm $(OBJECTS)
```

2.2 Pasta exp

Listing 3: definition.h

```
#pragma once
#include <utility>
#define FAILURE false
#define SUCCESS true

using AtributeValue = int;
using Priority = int;
using Token_name = int;
using Token_type =
    enum : int
    {
        endExpression,
        leftParen,
        rightParen,
        unaryOp,
        binaryOp,
        operand,
        number
    };

using Token = std::pair<Token_name,AtributeValue>;
```

Listing 4: expression.h

```
#pragma once
#include <queue>
#include <list>
#include <map>
#include <string>
#include <utility>
#include "definitions.h"
#include "lexicon.h"

using ErrorCode = bool;

class Expression
{
    std::list<Token> tokenized_expr;
    Lexicon symbol_table;
    public:
        Expression();
        Expression(std::string &expression); //Tokenize the expression
        ErrorCode infixToPostfix(); //certify that expression is valid first
        ErrorCode evaluateAt(double x, double &f_of_x);
        void tokenizeExpression(std::string &expression);
        void getIteratorRange(std::list<Token>::iterator &start,
                              std::list<Token>::iterator &end);
        ErrorCode isValid(); //check for infix
    private:
        void removeFirstToken(); //move back to private
        void addToken(Token &new_token);
```

```
        float do_unary(double x, Token_name type);
        float do_binary(double x, double y, Token_name type);
};

/*Autenticate lexical correctness of expression before sending to the class*/
/*Also add whitespace to better identify lexemes*/
ErrorCode addSpaces(std::string &expression);
```

Listing 5: token.h

```
#pragma once
#include "definitions.h"
#include <string>

struct Token_data
{
    std::string name;
    double value;
    Priority priority;
    Token_data(std::string token_name, double value, Priority priority);
};
```

Listing 6: lexicon.h

```
#include <map>
#include <string>
#include <vector>
#include "definitions.h"
#include "token.h"
#define NON_EXISTENT -1

class Lexicon
{
    //shall only be used to setup expression
    std::map<std::string, AtributeValue> lexeme_map;
    std::vector<Token_data*> symbol_table;
    public:
        Lexicon() = default;
        void setStandardTokens(); //sets up the map to lexemes
        Token_data* getTokenInfo(AtributeValue token_id);
        AtributeValue newToken(Token_data *new_token);
        AtributeValue findAtribute(std::string &lexeme);
};
```

Listing 7: expression.cpp

```
#include <cmath>
#include <cstdio>
#include <math.h>
```

```cpp
#include <string>
#include <stack>
#include <iostream>
#include <cmath>
#include <utility>
#include "definitions.h"
#include "expression.h"

inline bool isAlphabetLexeme(std::string &lexeme)
{
   return lexeme == "x" || lexeme == "e" || lexeme == "pi" ||
          lexeme == "sin" || lexeme == "cos" || lexeme == "exp" ||
          lexeme == "ln" || lexeme == "lg" || lexeme == "abs" ||
          lexeme == "sqrt" || lexeme == "arctan";
}

ErrorCode addSpaces(std::string &expression)
{
    ErrorCode lexicalError = false;
    std::string spacedExpression, auxStr;
    std::string opLexemes = ";()+-*/^";
    auto iteratorString = expression.begin();

    while (iteratorString != expression.end() && !lexicalError)
    {
        if(*iteratorString == ' ')
            iteratorString++;
        else if(*iteratorString >= 'a' && *iteratorString <= 'z')
        {
            do
            {
                auxStr += *iteratorString;
                iteratorString++;
            }while (*iteratorString >= 'a' && *iteratorString <= 'z');

            if(isAlphabetLexeme(auxStr))
                spacedExpression += auxStr + ' ';
            else
                lexicalError = true;
            auxStr.clear();
        }
        else if(*iteratorString >= '1' && *iteratorString <= '9')
        {
            do
            {
                auxStr += *iteratorString;
                iteratorString++;
            }while (*iteratorString >= '0' && *iteratorString <= '9');
            spacedExpression += auxStr + ' ';
            auxStr.clear();

            /* check for float expression */
            /* 3.8*x^3.35 */
        }
        else
        {
            auto auxIterator = opLexemes.begin();
            while (*auxIterator != *iteratorString && auxIterator != opLexemes.end())
                auxIterator++;
            if(auxIterator == opLexemes.end())
```

```
64              {
65                  lexicalError = true;
66                  std::cout << "Error with operation\n";
67                  std::cout << *iteratorString << '\n';
68              }
69              else
70              {
71                  spacedExpression += *iteratorString;
72                  if(*iteratorString == ';')
73                      iteratorString = expression.end();
74                  else
75                  {
76                      iteratorString++;
77                      spacedExpression += ' ';
78                  }
79              }
80          }
81      }
82
83      expression.clear();
84      expression = spacedExpression;
85
86      return lexicalError;
87  }
88
89  Expression::Expression(std::string &expression)
90  {
91      symbol_table.setStandardTokens();
92      tokenizeExpression(expression);
93  }
94
95  Expression::Expression()
96  {
97      symbol_table.setStandardTokens();
98  }
99
100 void Expression::tokenizeExpression(std::string &expression)
101 {
102     auto iteratorExpr = expression.begin();
103     bool leading = true;
104     std::string auxStr;
105     AtributeValue token_id;
106     Token new_token;
107     while(iteratorExpr != expression.end())
108     {
109         if(*iteratorExpr == ' ')
110             iteratorExpr++;
111         if(*iteratorExpr == '+' && leading)
112             iteratorExpr += 2;
113         else if(*iteratorExpr == '-' && leading)
114             *iteratorExpr = '~';
115
116         do
117         {
118             auxStr += *iteratorExpr;
119             iteratorExpr++;
120         }while(*iteratorExpr != ' ' && iteratorExpr != expression.end());
121
122         //std::cout << auxStr << '\n';
123         //std::cout << auxStr[0] << '\n';
```

```
124         //if(iteratorExpr == expression.end())
125         // std::cout << "Iterator got to the end\n";
126 
127         if(auxStr[0] >= 'a' && auxStr[0] <= 'z')
128         {
129             if(auxStr == "x")
130                 new_token.first = operand;
131             else if (auxStr == "e" || auxStr == "pi")
132                 new_token.first = number;
133             else
134                 new_token.first = unaryOp;
135         }
136         else if(auxStr[0] >= '0' && auxStr[0] <= '9')
137             new_token.first = number;
138         else
139         {
140             if(auxStr == "(")
141                 new_token.first = leftParen;
142             else if(auxStr == ")")
143                 new_token.first = rightParen;
144             else if(auxStr == ";")
145                 new_token.first = endExpression;
146             else if(auxStr == "~")
147                 new_token.first = unaryOp;
148             else
149                 new_token.first = binaryOp;
150         }
151 
152         if(new_token.first == leftParen || new_token.first == unaryOp ||
153            new_token.first == binaryOp)
154             leading = true;
155         else
156             leading = false;
157 
158         token_id = symbol_table.findAtribute(auxStr);
159         if(token_id == NON_EXISTENT)
160         {
161             //std::cout << "float -> " << auxStr << '\n';
162             float value = std::stof(auxStr);
163             token_id = symbol_table.newToken(new Token_data(auxStr,value,0));
164         }
165         auxStr.clear();
166         new_token.second = token_id;
167         addToken(new_token);
168     }
169 }
170 
171 ErrorCode Expression::infixToPostfix()
172 {
173     std::stack<Token> delay_ops;
174     Token current, prior;
175     auto iterInfix = tokenized_expr.begin();
176     while(iterInfix->first != endExpression)
177     {
178         //std::cout << "somgoidfg\n";
179         switch (iterInfix->first)
180         {
181             case operand:
182             case number:
183                 addToken(*iterInfix);
```

```
                break;
            case leftParen:
                delay_ops.push(*iterInfix);
                break;
            case rightParen:
                prior = delay_ops.top();
                while (prior.first != leftParen)
                {
                    addToken(prior);
                    delay_ops.pop();
                    prior = delay_ops.top();
                }
                delay_ops.pop();
                break;
            case unaryOp:
            case binaryOp:
                bool end_right = false;
                do
                {
                    if(delay_ops.empty())
                        end_right = true;
                    else
                    {
                        prior = delay_ops.top();
                        if(prior.first == leftParen)
                            end_right = true;
                        else if(symbol_table.getTokenInfo(prior.second)->priority <
                                symbol_table.getTokenInfo(iterInfix->second)->priority)
                            end_right = true;
                        else if(symbol_table.getTokenInfo(iterInfix->second)->priority == 6)
                            end_right = true;
                        else
                            addToken(prior);
                        if(!end_right)
                            delay_ops.pop();
                    }
                }while (!end_right);
                delay_ops.push(*iterInfix);
                break;
        }

        iterInfix++;
        removeFirstToken();
        //for (auto iter : tokenized_expr)
        // std::cout << iter.second << ' ';
        //std::cout<<'\n';
    }

    while(!delay_ops.empty())
    {
        prior = delay_ops.top();
        addToken(prior);
        delay_ops.pop();
    }
    prior = tokenized_expr.front();
    removeFirstToken();
    addToken(prior);

    return SUCCESS;
}
```

```
244
245 ErrorCode Expression::evaluateAt(double x,double &f_of_x)
246 {
247     std::string lexeme_x = "x";
248     symbol_table.getTokenInfo(symbol_table.findAtribute(lexeme_x))->value = x;
249     double first_elem, second_elem;
250     std::stack<double> operands;
251     auto iterExpr = tokenized_expr.begin();
252     do
253     {
254         switch (iterExpr->first)
255         {
256             case unaryOp:
257                 if(operands.empty())
258                     return FAILURE;
259                 first_elem = operands.top();
260                 operands.pop();
261                 operands.push(do_unary(first_elem, iterExpr->second));
262                 break;
263             case binaryOp:
264                 if(operands.empty())
265                     return FAILURE;
266                 second_elem = operands.top();
267                 operands.pop();
268                 if(operands.empty())
269                     return FAILURE;
270                 first_elem = operands.top();
271                 operands.pop();
272                 operands.push(do_binary(first_elem,second_elem,iterExpr->second));
273                 break;
274             case operand:
275             case number:
276                 operands.push(symbol_table.getTokenInfo(iterExpr->second)->value);
277                 break;
278             case endExpression:
279                 break;
280         }
281         //std::cout << operands.top() << '\n';
282         iterExpr++;
283     }while (iterExpr->first != endExpression);
284
285     if(operands.empty())
286         return FAILURE;
287     f_of_x = operands.top();
288     operands.pop();
289     if(!operands.empty())
290         return FAILURE;
291     return SUCCESS;
292 }
293
294 void Expression::addToken(Token &new_token)
295 {
296     tokenized_expr.push_back(new_token);
297 }
298
299 void Expression::removeFirstToken()
300 {
301     tokenized_expr.pop_front();
302 }
303
```

```
304 ErrorCode Expression::isValid()
305 {
306     auto iterToken = tokenized_expr.begin();
307     int parent_count = 0;
308     bool leading = true;
309     Token_name type;
310     while (iterToken->first != endExpression)
311     {
312         type = iterToken->first;
313         //std::cout << "Type -> " << type << '\n';
314         if(type == rightParen || type == binaryOp)
315         {
316             if(leading)
317                 return FAILURE;
318         }
319         else if(!leading)
320             return FAILURE;
321 
322         if(type == leftParen)
323             parent_count++;
324         else if(type == rightParen)
325         {
326             parent_count--;
327             if(parent_count < 0)
328                 return FAILURE;
329         }
330         if(type == binaryOp || type == unaryOp || type == leftParen)
331             leading = true;
332         else
333             leading = false;
334 
335         iterToken++;
336     }
337 
338     if(leading)
339     {
340         //std::cout << "\nLeading FAILURE\n";
341         return FAILURE;
342     }
343     if(parent_count > 0)
344     {
345         //std::cout << "\nparent_count FAILURE\n";
346         return FAILURE;
347     }
348 
349     return SUCCESS;
350 }
351 
352 void Expression::getIteratorRange(std::list<Token>::iterator &start,
353                                   std::list<Token>::iterator &end)
354 {
355     start = tokenized_expr.begin();
356     end = tokenized_expr.end();
357 }
358 
359 float Expression::do_unary(double x, Token_name type)
360 {
361     switch (type)
362     {
363         case 3:
```

```
            return -x;
        case 4:
            return std::abs(x);
        case 5:
            return std::sqrt(x);
        case 6:
            return std::exp(x);
        case 7:
            return std::log(x);
        case 8:
            return std::log2(x);
        case 9:
            return std::sin(x);
        case 10:
            return std::cos(x);
        case 11:
            return std::atan(x);
        default:
            break;
    }
    return 0.0f;
}

float Expression::do_binary(double x, double y, Token_name type)
{
    switch (type)
    {
        case 12:
            return x + y;
        case 13:
            return x - y;
        case 14:
            return x * y;
        case 15:
            return x / y;
        case 16:
            return std::pow(x, y);
    }

    return 0.0f;
}
```

Listing 8: lexicon.cpp

```
#include "token.h"
#include <string>

Token_data::Token_data(std::string token_name, double value, Priority priority)
{
   this->name = token_name;
   this->value = value;
   this->priority = priority;
}
```

Listing 9: lexicon.cpp

```
#include "lexicon.h"
#include "definitions.h"
#include <cmath>
#include <math.h>

void Lexicon::setStandardTokens()
{
   lexeme_map =
   {
      {";",0},
      {"(",1},
      {")",2},
      {"~",3},
      {"abs",4},
      {"sqrt",5},
      {"exp",6},
      {"ln",7},
      {"lg",8},
      {"sin",9},
      {"cos",10},
      {"arctan",11},
      {"+",12},
      {"-",13},
      {"*",14},
      {"/",15},
      {"^",16},
      {"x",17},
      {"pi",18},
      {"e",19}
   };

   symbol_table =
   {
      new Token_data(";",0.0,6),
      new Token_data("(",0.0,6),
      new Token_data(")",0.0,6),
      new Token_data("~",0.0,6),
      new Token_data("abs",0.0,6),
      new Token_data("sqrt",0.0,6),
      new Token_data("exp",0.0,6),
      new Token_data("ln",0.0,6),
      new Token_data("lg",0.0,6),
      new Token_data("sin",0.0,6),
      new Token_data("cos",0.0,6),
      new Token_data("arctan",0.0,6),
      new Token_data("+",0.0,4),
      new Token_data("-",0.0,4),
      new Token_data("*",0.0,5),
      new Token_data("/",0.0,5),
      new Token_data("^",0.0,6),
      new Token_data("x",0.0,0),
      new Token_data("pi",M_PI,0),
      new Token_data("e",M_E,0),
   };
}

AtributeValue Lexicon::findAtribute(std::string &lexeme)
{
   if (lexeme_map.find(lexeme) != lexeme_map.end())
```

```
60        return lexeme_map.find(lexeme)->second;
61     else
62        return NON_EXISTENT;
63  }
64
65  AtributeValue Lexicon::newToken(Token_data *new_token)
66  {
67     symbol_table.push_back(new_token);
68     return symbol_table.size() - 1;
69  }
70
71  Token_data* Lexicon::getTokenInfo(AtributeValue token_id)
72  {
73     if(token_id < symbol_table.size())
74        return symbol_table[token_id];
75     else
76        return nullptr;
77  }
```

2.3 Pasta integral

Listing 10: integral.h

```
#include "../exp/expression.h"
#include <utility>
#include <vector>

struct TrapezoidIntegral
{
    int nOfTrapz;
    int precision;
    double errorRounding;
    double sumTraps;
    double x_start, x_end;
    void calculateIntegral(Expression &expr,
        std::vector<std::pair<double, double>> &fnTable);
    TrapezoidIntegral() = default;
    TrapezoidIntegral(double start, double end, int num, int precision);
};
```

Listing 11: integral.cpp

```
#include "integral.h"
#include <cmath>
#include <math.h>
#include <utility>
#include <vector>

void setNumber(int precision, double &value);

TrapezoidIntegral::TrapezoidIntegral(double start, double end,
                                int num, int precision)
{
    x_start = start;
    x_end = end;
    nOfTrapz = num;
    this->precision = precision;
}

void TrapezoidIntegral::calculateIntegral(Expression &expr,
        std::vector<std::pair<double, double>> &fnTable)
{
    double increment = std::abs(x_end - x_start);
    increment /= static_cast<double>(nOfTrapz);
    double x = x_start;
    double f_late, f_early;
    fnTable.clear();
    sumTraps = 0.0F;

    expr.evaluateAt(x, f_late);
    setNumber(precision,f_late);
    fnTable.push_back(std::pair<double, double>(x, f_late));
    x += increment;
    expr.evaluateAt(x, f_early);
```

```
    setNumber(precision, f_early);

    for (int i = 1; i <= nOfTrapz; i++)
    {
        fnTable.push_back(std::pair<double, double>(x, f_early));
        x += increment;
        sumTraps += increment * (f_early + f_late) / 2.0F;
        f_late = f_early;
        expr.evaluateAt(x, f_early);
        setNumber(precision, f_early);
    }

    errorRounding = nOfTrapz *
                (5.0F / std::pow(10.0F, precision + 1)) * increment;
}

void setNumber(int precision, double &value)
{
    value *= pow(10.0, precision);
    value = std::round(value);
    value /= pow(10.0, precision);
}
```

3 Exemplo

A seguir temos um screenshot demostrando um exemplo do programa em execução, na qual este avalia a integral $\int_{1.5}^{2.5} \frac{e^{-\frac{x^2}{2}}}{2\pi} dx$, sendo que o número de casas decimais e 5 é e a quantidade de trapézios é 10.

```
Escreva uma expressao:
f(x) = e^(-x^2/2)/sqrt(2*pi)


Escreva o ponto extremo da esquerda:
1.5
Escreva o ponto extremo da direita:
2.5

Escreva o numero de casas decimais
5
Escreva o numero de trapezios:
10


Saida:

Erro de arredondamento = 0.000005
Valor da Integral = 0.06072
Tabela de Valores:
x       | f(x)
1.50000 | 0.12952
1.60000 | 0.11092
1.70000 | 0.09405
1.80000 | 0.07895
1.90000 | 0.06562
2.00000 | 0.05399
2.10000 | 0.04398
2.20000 | 0.03547
2.30000 | 0.02833
2.40000 | 0.02239
2.50000 | 0.01753
Deseja sair do programa?
Sim (Escreva q)  Nao (Escreva qualquer coisa)
```