Федеральное государственное автономное образовательное учреждение

высшего образования

«Нижегородский государственный университет

им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Разбор и вычисление символьных математических выражений с использованием стека

Выполнил:

студент и-та ИТММ гр. 0823-1

Конаков А.В.

Проверил:

ассистент каф. ПИнж, ИИТММ

Сиднев А.А.

Нижний Новгород

2016 г.

Содержание

[Введение 3](#__RefHeading___Toc270962758)

[Постановка задачи 4](#__RefHeading___Toc270962759)

[Руководство пользователя 5](#__RefHeading___Toc270962760)

[Руководство программиста 6](#__RefHeading___Toc270962761)

[Описание структур данных 6](#__RefHeading___Toc270962762)

[Описание алгоритмов 6](#__RefHeading___Toc270962763)

[Описание структуры программы 6](#__RefHeading___Toc270962764)

[Заключение 7](#__RefHeading___Toc270962765)

[Литература 8](#__RefHeading___Toc270962766)

[Приложения 9](#__RefHeading___Toc270962767)

[Приложение 1 9](#__RefHeading___Toc270962768)

[Приложение 2 9](#__RefHeading___Toc270962769)

# Введение

Выполнение данной лабораторной работы позволило мне создать основу гибкого и расширяемого интерфейса класса разбора и вычисления математических выражений с учётом приоритета операций, скобок и функций; также проделанная работа позволила освоить некоторые базовые функции библиотеки компьютерного зрения OpenCV(создание пустых изображений и отрисовка простейших графических примитивов, таких как точки и прямые).

# Постановка задачи

Необходимо реализовать программу, выполняющую разбор, обработку математических выражений и их вычисление.

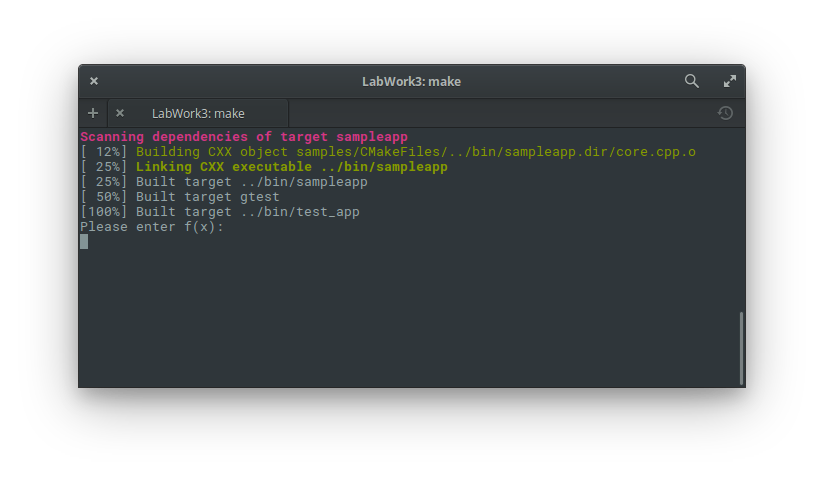
Программа должна обеспечивать:

1. Возможность принимать на ввод арифметически корректные математические выражения содержащие:
   1. Знаки основных арифметических операций (**+, -, -(***унарный***), \*, /**),
   2. Операцию возведения в степень, функций(в частности функции взятия модуля от выражения и др.),
   3. Констант(целочисленного или вещественного типа), в том числе **℮** и ***π,***
   4. Переменные любой допустимой длины, начинающиеся либо с букв латинского алфавита, символа нижнего подчёркивания или знака «$», затем допустимы также цифры,
2. Обрабатывать базовый набор возможных ошибок в арифметическом выражении:
   1. Переизбыток скобок,
   2. Проверка на корректность имён переменных(несоответствующим правилам именования переменных описанных в пункте 1),
   3. Проверка на отсутствие операций(например между переменной и открывающей скобкой),
3. Простейшую визуальную составляющую, на примере прорисовки графика введённой непосредственно в программе функции.

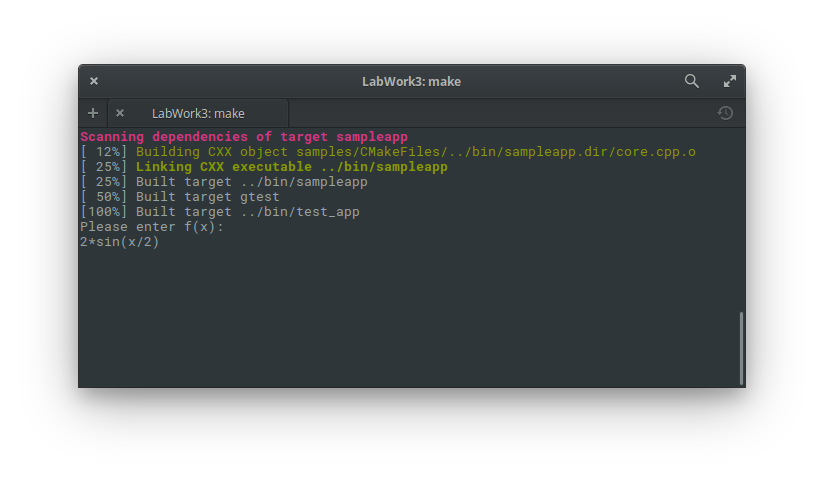
# Руководство пользователя

Получим график функции .

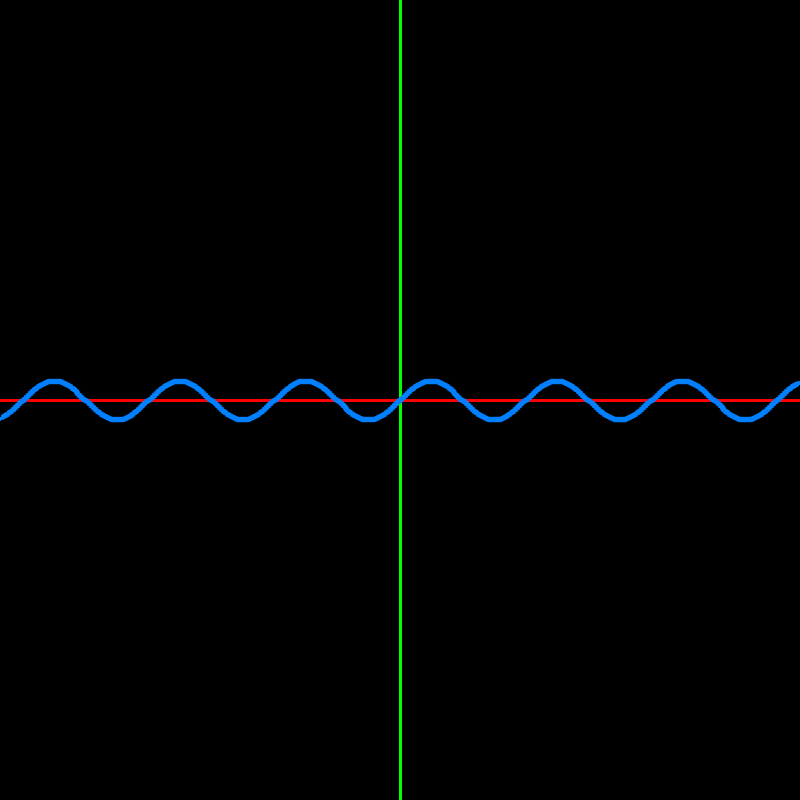
1. Для этого запустим программу(на Linux ***./bin/sampleapp***):



1. Введём функцию на приглашение консоли : ***2\*sin(x/2)*** и нажмём <Enter>:



1. Получаем результат:



# Руководство программиста

## Описание структуры программы

Программа состоит из 5 файлов исходного кода:

* *MathExpr.hpp*
  + *Содержит:*
    - *класс MathExpr*
    - *класс MathExpr::ExprException*
* *LexUtility.hpp*
  + *Содержит*
    - *структуру Variable*
    - *перечисление LexType*
    - *функцию isVariable*
    - *функцию itos*
    - *функцию isHigherPriority*
    - *функцию isFunction*
    - *функцию detectTokType*
    - *функцию calculateFunc*
* *core.cpp*
* *test\_expression.cpp*

*Программа работает следующим образом:*

* *После запуска пользователю предоставляется возможность ввести выражение*
* *После ввода пользователь получает результат график функции в отдельном окне*

*Работа с главным классом MathExpr состоит из шагов:*

* *Создание экземпляра класса*
* *Загрузка строки соответствующего математическому выражению для расчёта с помощью метода loadExpr,*
* *(По необходимости) Загрузка значений переменных, используемых в выражений с помощью метода loadVariable,*
* *Исполнение выражения или просто его перевод(например если не были загружены значения переменных) при помощи методов compute или translate соответственно.*

## Описание структур данных

* *struct Variable -* вспомогательная структура для хранения связи между названием переменной и её значением,
* enum LexType - вспомогательное перечисление для определения типов лексем,
* *string functions[]* - массив хранящий названия функций, которые поддерживает программа,
* *string priorityList[]* - массив хранящий названия функций в порядке приоритета(чем меньше индекс, тем выше приоритет),
* *string itos(unsigned int)* - функция преобразующая целое в строку,
* *bool isHigherPriority(const string &, const string &)* - функция для сравнения приоритета левой и правой операций (true, если аргумент слева имеет приоритет выше),
* *bool isVariable(char x)* - вспомогательная функция для оценки первого символа на его соответствию правилам именования переменных,
* LexType detectTokType(string tok) — функция для определения типа подвыражения возвращает перечисление соответствующего значения

## Описание алгоритмов

Основными в программе являются два алгоритма:

1. Алгоритм перевода математического выражения в обратную польскую запись:
   1. Для упрощения сначала наполняем очередь разбивая данное нам выражение на подвыражения взяв за разделители все бинарные операторы и скобки,
   2. Создаём два стека: стек операций и стек значений,
   3. Повторяем выталкивание из «заведённой» на шаге 1.1 очереди пока она не будет опустошена, выталкиваемый таким образом элемент оценивается его тип: открывающая/закрывающая скобка, операция(в т.ч. функция) или значение(любое в т.ч. любая переменная):
      1. Открывающая скобка - заталкивается в стек операций,
      2. Значение - отправляем в стек значений,
      3. Встречая закрывающую скобку - выполняем все операции из стека операций пока не встретится открывающая скобка,
      4. Операция — если приоритет текущей операции выше операции на вершине стека то выполняем операцию из стека операций, иначе просто заталкиваем текущую операцию в стек.
2. Алгоритм вычисления выражения математического выражения в обратной польской записи:
   1. «Заталкиваем» в стек лексемы из переведённого алгоритмом выше выражения:
      1. Константы заталкиваются в него сразу,
      2. Переменные заменяются соответствующими им значениями и также заталкиваются в стек,
      3. Операции выполняются непосредственно с выталкиванием из стека необходимого количества констант.

# Заключение

В лабораторной работе был написан класс для расчёта математических выражений, а также графическая часть реализующая его возможности в форме графиков от введённых выражений.

# Литература

1. Обратная польская запись - [https://ru.wikipedia.org/wiki/Обратная\_польская\_запись]
2. Справочник по C++ - [http://ru.cppreference.com/w/]

# Приложения

## Приложение 1(MathExpr.hpp)

// Copyright 2016, ver. 0.001 by AKonia

#ifndef \_\_MATHEXPR\_HPP\_\_

#define \_\_MATHEXPR\_HPP\_\_

#include "LexUtility.hpp"

#include <stack>

#include <queue>

#include <vector>

#include <string>

#include <cstring>

#include <cmath>

using std::stack;

using std::string;

using std::vector;

using std::queue;

class MathExpr

{

public: // Обслуживание

class ExprException

{

string fullMsg;

public:

ExprException(string msg, int line)

{

fullMsg = "On line " + itos(line) + " were thrown exception: " + msg;

}

string getMessage()

{

return fullMsg;

}

};

private: // Данные с которыми работает вид

string expr;

bool exprValidationStatus;

bool translated;

bool isTranslated;

//vector

vector<Variable> variables;

vector<bool> isConstant;

public: // Основные функции, с помощью которых происходит взаимотействие из вне

MathExpr()

{

translated = false;

addConstant("e", 2.71828);

#ifdef M\_PI

addConstant("pi", M\_PI);

#else

addConstant("pi", 3.14159);

#endif

}

string getExpr()

{

return expr;

}

void loadExpr(const string &expr);

void loadVariable(string name, double value)

{

for(int i = 0; i < variables.size(); i++)

if(name == variables[i].name)

if(!isConstant[i])

{

variables[i].value = value;

return;

}

else

throw ExprException("Impossible to write variable value, it conflicting with constant name: '" + name + "'", \_\_LINE\_\_);

addVariable(name, value);

}

void translate();

double compute();

bool isValid();

private:

double getValueByName(string name)

{

for(int i = 0; i < variables.size(); i++)

if(variables[i].name == name)

return variables[i].value;

throw ExprException("Can't find variable '" + name + "' value, possible it wasn't entered", \_\_LINE\_\_);

}

void addConstant(string name, double value)

{

if(detectTokType(name) == VARIABLE)

{

Variable var;

var.name = name;

var.value = value;

variables.push\_back(var);

isConstant.push\_back(true);

}

else

throw ExprException("Can't add value because it isn't supported format of the variable", \_\_LINE\_\_);

}

void addVariable(string name, double value)

{

addConstant(name, value);

isConstant[isConstant.size()-1] = false;

}

};

void MathExpr::loadExpr(const string &expr)

{

translated = false;

if(expr != "")

this->expr = expr;

else

exprValidationStatus = false;

}

void MathExpr::translate()

{

if(translated)

return;

expr = "(" + expr + ")";

queue<string> tokens;

string delimSequence = "";

int wasDelim = 1;

for(int i = 0; i < expr.length(); i++)

{

if(isspace(expr[i]))

continue;

if(strchr("+-\*/()^|", expr[i]))

{

if(!wasDelim)

{

tokens.push(delimSequence);

delimSequence = "";

}

wasDelim = 1;

delimSequence += expr[i];

tokens.push(delimSequence);

delimSequence = "";

}

else

{

wasDelim = 0;

delimSequence += expr[i];

}

}

stack<string> exprOperators, values;

LexType currType = UNKNOWN\_TYPE,

lastType = UNKNOWN\_TYPE;

do

{

currType = detectTokType(tokens.front()); // получили текущую лексему

//cout << "Current token is: " << tokens.front() << endl;

switch(currType) // в соответствии с типом лексемы выбираем что делать

{

case OPERATION:

if((lastType == OPEN\_BRACKET || lastType == UNKNOWN\_TYPE) &&

(tokens.front() == "-" || tokens.front() == "+"))

values.push(string("0"));

else if(lastType != VARIABLE && lastType != CLOSE\_BRACKET && !isFunction(tokens.front()))

throw ExprException("Unexpected expression", \_\_LINE\_\_);

if(!exprOperators.empty() && isHigherPriority(exprOperators.top(), tokens.front())) // если предыдущая операция имеет больший приоритет, то исполняем её

{

if(isFunction(exprOperators.top()))

{

string temporary = values.top();

values.pop();

values.push(temporary + " " + exprOperators.top());

}

else

{

string right = values.top();

values.pop();

string left = values.top();

values.pop();

values.push(left + " " + right + " " + exprOperators.top());

}

exprOperators.pop();

}

lastType = OPERATION;

exprOperators.push(tokens.front());

break;

case OPEN\_BRACKET:

if(lastType == VARIABLE)

throw ExprException("Bad expression: you can't miss operations before opening braces", \_\_LINE\_\_);

lastType = OPEN\_BRACKET;

exprOperators.push(tokens.front());

break;

case CLOSE\_BRACKET:

lastType = CLOSE\_BRACKET;

if(exprOperators.empty())

throw ExprException("No operators before CLOSE\_BRACKET, possibly too much brackets",\_\_LINE\_\_);

while(detectTokType(exprOperators.top()) != OPEN\_BRACKET)

{

if(isFunction(exprOperators.top()))

{

string temporary = values.top();

values.pop();

values.push(temporary + " " + exprOperators.top());

}

else

{

string right = values.top();

values.pop();

string left = values.top();

values.pop();

values.push(left + " " + right + " " + exprOperators.top());

}

exprOperators.pop();

}

exprOperators.pop();

break;

case UNKNOWN\_TYPE:

throw ExprException("UNKNOWN\_TYPE of lexem in expression", \_\_LINE\_\_);

break;

default:

if(lastType == VARIABLE)

throw ExprException("NOT CORRECT DATA: VARIABLE AFTER VARIABLE", \_\_LINE\_\_);

lastType = VARIABLE;

values.push(tokens.front());

break;

}

tokens.pop();

}

while(!tokens.empty());

while(!exprOperators.empty() && !values.empty())

{

if(isFunction(exprOperators.top()))

{

string temporary = values.top();

values.pop();

values.push(temporary + " " + exprOperators.top());

}

else

{

string right = values.top();

values.pop();

string left = values.top();

values.pop();

values.push(left + " " + right + " " + exprOperators.top());

}

exprOperators.pop();

}

expr = values.top();

translated = true;

exprValidationStatus = true;

}

double MathExpr::compute()

{

if(exprValidationStatus == false)

{

if(!translated)

translate();

if(exprValidationStatus == false)

throw ExprException("Expression is not valid for computing", \_\_LINE\_\_);

}

queue<string> tokens;

string delimSequence = "";

int wasDelim = 1;

for(int i = 0; i < expr.length(); i++)

{

if(isspace(expr[i]))

{

if(delimSequence == "")

continue;

tokens.push(delimSequence);

delimSequence = "";

}

else

{

delimSequence += expr[i];

}

}

if(delimSequence != "")

tokens.push(delimSequence);

stack<double> preValues;

char \*\* passive = 0;

double value = 0.0;

while(!tokens.empty())

{

//cout << tokens.front() << endl;

LexType currType = detectTokType(tokens.front());

switch(currType)

{

case OPERATION:

if(isFunction(tokens.front()))

{

double singleValue = preValues.top();

preValues.pop();

preValues.push(calculateFunc(tokens.front(), singleValue, 0));

}

else

{

double lhVal, rhVal;

rhVal = preValues.top();

preValues.pop();

lhVal = preValues.top();

preValues.pop();

preValues.push(calculateFunc(tokens.front(), lhVal, rhVal));

}

break;

case INT\_NUMBER:

case REAL\_NUMBER:

value = strtod(tokens.front().c\_str(), passive);

preValues.push(value);

break;

case UNKNOWN\_TYPE:

throw ExprException("UNKNOWN\_TYPE of expression", \_\_LINE\_\_);

break;

case CONSTANT:

case VARIABLE:

preValues.push(getValueByName(tokens.front()));

break;

}

tokens.pop();

}

return preValues.top();

}

bool MathExpr::isValid()

{

return exprValidationStatus;

}

#endif // \_\_MATHEXPR\_HPP\_\_

## Приложение 2(LexUtility.hpp)

#include <cctype>

#include <string>

#include <cstring>

#include <cmath>

#include <cstdlib>

using std::string;

const int qOfSupportedFunctions = 6;

const int qOfPriority = qOfSupportedFunctions + 5;

const char \* operations = "+-\*/%^";

const char \* delims = "(){}[]|+-\*/%^";

const char \* openingBrackets = "([{|";

const char \* closingBrackets = ")]}|";

string itos(unsigned int);

bool isVariable(char x);

bool isHigherPriority(const string &lhs, const string &rhs);

bool isFunction(string operation);

string functions[qOfSupportedFunctions] =

{

string("sin"),

string("cos"),

string("tan"),

string("log"),

string("abs")

};

string priorityList[qOfPriority] =

{

string("sin"),

string("cos"),

string("tan"),

string("log"),

string("abs"),

string("^"),

string("/"),

string("\*"),

string("-"),

string("+")

};

struct Variable

{

string name;

float value;

};

enum LexType

{

UNKNOWN\_TYPE = 0,

VARIABLE,

REAL\_NUMBER,

INT\_NUMBER,

CONSTANT,

OPERATION,

OPEN\_BRACKET,

CLOSE\_BRACKET

};

bool isVariable(char x)

{

return (isalnum(x) || (x == '\_') || (x == '$'));

}

string itos(unsigned int number)

{

string result = "";

string reversed = "";

while(number != 0)

{

reversed += (char)('0' + number % 10);

number /= 10;

}

for(int i = 0; i < reversed.length(); i++)

result += reversed[i];

return result;

}

bool isHigherPriority(const string & lhs, const string & rhs)

{

for(int i = 0; i < qOfPriority; i++)

{

if(priorityList[i] == lhs)

return true;

if(priorityList[i] == rhs)

return false;

}

}

bool isFunction(string operation)

{

for(int i = 0; i < qOfSupportedFunctions; i++)

if(operation == functions[i])

return true;

return false;

}

LexType detectTokType(string tok)

{

size\_t toklen = tok.length();

if(toklen < 1)

return UNKNOWN\_TYPE;

if(toklen == 1)

{

if(strchr(operations, tok[0]))

return OPERATION;

if(strchr(openingBrackets, tok[0]))

return OPEN\_BRACKET;

if(strchr(closingBrackets, tok[0]))

return CLOSE\_BRACKET;

}

if(isdigit(tok[0]))

{

LexType numericType = INT\_NUMBER;

for(int i = 1; i < toklen; i++)

if(!isdigit(tok[i]))

{

if(!((tok[i] == '.') || (tok[i] == ',')) || REAL\_NUMBER == numericType)

return UNKNOWN\_TYPE;

else

numericType = REAL\_NUMBER;

}

return numericType;

}

if(toklen == 3)

for(int i = 0; i < 5; i++)

if(tok == functions[i])

return OPERATION;

if(isalpha(tok[0]) || tok[0] == '\_' || tok[0] == '$')

{

for(int i = 1; i < toklen; i++)

if(!isVariable(tok[0]))

return UNKNOWN\_TYPE;

return VARIABLE;

}

return UNKNOWN\_TYPE;

}

bool findValByName(string name, double &value)

{

value = 1.0;

return true;

}

double calculateFunc(string funcName, double l, double r)

{

if(!isFunction(funcName))

{

switch(funcName[0])

{

case '+':

return l+r;

case '-':

return l-r;

case '\*':

return l\*r;

case '/':

return l / r;

case '^':

return pow(l, r);

}

}

else

{

if(funcName == "sin")

return sin(l);

if(funcName == "cos")

return cos(l);

if(funcName == "abs")

return abs(l);

if(funcName == "log")

return log(l);

if(funcName == "tan")

return tan(l);

}

}