Mathematical Equations solver and parser

By

Elsayed Mohamed Elsayed Awdallah

Mohamed Ahmed Abd El Moneim Elbagoury

Moustafa Yasser Abozeid Elsayed

A Graduation Project Documentation Submitted to
The Faculty of Computers & Information Zagazig University

CS Department

Table of Contents

Chapter 1

Introduction

Introduction

The abstract concept of the project "Mathematical Equation Parser and Solver" is to take a system of mathematical equations or an expression and parse it into a matrix then solve it.

As an application we created an application that can receive user input in various formats:

- Entering input using standard text box.
- A text file containing a group of equations or expressions.
- An image file that is scanned and then converted to text using OCR.

The application then parses and solves the system using the solver and prints back the result to the end user.

Problem overview

To develop a software to solve a mathematical equations you need to:

- Define regular expressions for the lexer.
- Define a grammar for mathematical equations and expressions.
- Model the mathematical parts of the equation: Terms, Factors and their operation.
- Implement the parser.
- Model the system into solvable format.
- Solve the system.
- Print back the system.

Evaluation of existing system

There exists some applications that help users who are interested in mathematics and it's purpose differs from one another ranging from teaching maths, solving mathematical problems, practicing, use in engineering purposes such as engineering projects.
In general there are many applications that are oriented to one or more than a mathematical field .

Math Mechanixs:
This program is for the serious mathematicians, advanced mathematics and advanced physics students. It is not a training aid or a spreadsheet program, rather it works using a Math Editor allowing the user to enter the mathematical expression. The software uses a multiple document interface so that you can work on multiple solutions simultaneously. There is a fully featured scientific calculator which includes a very useful integrated variables and functions list window so that you can easily track defined variables and functions.

Octave:
Octave is a high-level language developed by the open source community which functions simply by providing a command line interface for solving linear and nonlinear numeric problems. For those who are familiar with MATLAB, they will have few problems picking up Octave because it works in conjunction with the former language.
The program basically features several very powerful tools for solving the kind of linear algebra equations that is extremely complex that would send shivers down the spine of even the most experienced mathematician. According to the developers, it also includes functions for manipulating polynomials and integrating differential and differential-algebraic equations if you know what those are in the first place! The one bonus for those familiar with other languages is that Octave can also be extended or used in

conjunction with modules written in C++, C, FORTRAN and others.

Microsoft Math:

Another software from the giant Microsoft Company. Microsoft Math gives students tools, tutorials and instructions, all in one place! Its large collection of tools, tutorials, and instructions helps students learn mathematical concepts, while quickly solving math and science problems. Students see how to solve problems step-by-step – instantly! Microsoft Math works for many grade levels, studying subjects from basic math to calculus.

So, it address students who are interested in learning Math.

Top features include:

- A full featured Graphing Calculator gives students extensive graphing and equation-solving capabilities to visualize problems and concepts.
- Step-by-Step Math Solutions that guide students through problems in subjects from pre-Algebra to calculus.
- The Formulas and Equations Library has more than 100 common math equations and formulas.
- The Triangle Solver is a graphing tool that students can use to explore triangles.

Project Goals

Purpose:

The main goal of the project is to create a cross-platform mathematical solver library that is able to parse and accurately solve mathematical equations, and use it in an application that receives input from users in different ways.

Advantage:

Provides a cross-platform library that can be used to create applications for Windows, Mac OS, Linux, and mobile platforms in C#.

Measurement:

The accuracy of solving and simplifying various types of mathematical equation systems and expressions.

Chapter 2

Lexical Analysis

The Lexer phase, lexical analyzer or scanner is the first phase in the program which take the equation as a plain text.

The main function of the lexer is to take a string of individual letters and divide them into terminal tokens based on a regular expression specification.

Lexers can be created from scratch or generated using one of the lexer generators.

For our project we picked the latter solution as it is:

- More reliable: a lot of project already depend on them.
- More efficient: most of generator optimize the resulting DFA for performance.
- Maintainable: project and generator are maintained separately.
- Easier: generators usually use smaller specification files, and easier notation.
- Lexer generator saves time and effort and reduces error.
- In handwritten program the program requires a fair amount of programmer time to create.

We picked the CSFlex Generator[http://csflex.sourceforge.net/] it's based on the infamous Flex[http://flex.sourceforge.net/] but generates C# and Java scanners' source files instead.

Being based on Flex, CSFlex is more reliable and easier than most of the other generator as Flex has a vast community support and tutorials.

Regular Expressions

It is an algebraic notation for describing sets of strings , the set of all integers constants or the set of all variables are sets of strings where the individual letters are taken from a particular alphabet,

such asset of strings is called the language of the algebraic notation that is compact and easy for humans to use and understand.

It is a simple language for denoting classes of strings. A regular expression is defined inductively over an alphabet with a set of basic operations.

An individual character stands for itself, except for the reserved characters `?*+|()^$/;.=<>[]{}"\`.

Precedence Rules

When we combine different constructor symbols In the regular expression, it is not clear how the different sub-expressions are grouped. So, there is the importance of precedence rules. For example the regular expression a|ab* we use precedence rules , similar to algebraic convention that 3+4*5 means 3 added to the product of 4 and 5 and not multiplying the sum of 3 and 4 by 5.

For regular expression, we use the following conventions : * binds tighter than concatenation which bends tighter than alternative (|).

Shorthand

Sequence of letters between square b represent the set of these letters example we use [ab01] awhile describing numbers, strings or variable names, we will often use shorthands for convenience. We introduce a shorthand for sets of letters.

As a shorthand for a|b|0|1. Additionally, we can interval notation to abbreviate [0123456789] to [0-9]. We can combine several intervals within one bracket. When using intervals, we must be aware of the ordering for the symbol involved

For example in our project we had to define a set of regular expressions for tokens this is a list of them:

Token Type	Regular Expression
ID	`[a-zA-Z][a-zA-Z0-9]*`
NUMBER	`([0-9]|[1-9][0-9]*)|([0-9]|[1-9][0-9]*)\.([0-9]|[1-9][0-9]*)`
Whitespaces	`[\r\t\n\f\ ]*`
Sin, Cos, Tan, Sinh, Cosh, Tanh...	`[Ss][Ii][Nn],[Cc][Oo][Ss],[Tt][Aa][Nn]...`
Signs	`\+, \-,\/,\*,\=...`
Parenthesises	`\(,\)`

Finite state automaton

If a language L is generated by a regular expression, then L is recognized by a DFA.

To transform regular expression into an efficient program, we use finite automaton.

We can transform regular expression to Non-deterministic finite automaton then transform it to a deterministic finite automaton or you can transform regular expression directly into deterministic automaton.

We transform regular expression into DFA by forming each sub-expression construct an DFA fragment and then combine these fragment into a bigger fragments. a fragment is not a complete DFA, so we complete construction by adding the necessary components to make a complete DFA . DFA function is to accept or reject a string .

A DFA fragment consists of number of states with transition between these states and if the string is accepted , the finite state should end in an state which is called accepted state (Note that there may be more than one accept state) and if the string ended up in an unaccepted state if the input or string is not accepted.

At the beginning of scanning a string the DFA starts from a state which is called the initial state and is marked by an arrow without a starting.

Lexer Generator

So, we will understand what is the lexer generator and what is it's hierarchy as we said before some lexers are constructed by the lexer generators.

Hierarchy and function:

Lexer Generator is a program that takes a specification as as input and generates a scanner or lexer source file as an output. Flex specification files consist of three basic regions.

Each section is separated from the others by a %% delimiter.

- User code.
- Options.
- Lexer rules.

User code:
The area in the lexer generator in which the code written left as is in the source file generated.
In this region we should specify our token types and anything we want to define before the lexer code.

For example:

```
using System;
using System.IO;

public enum TokenType{ ADD, SUB, DEV, MULTI, POW,
FACT, LPARENT, RPARENT, EQUAL, SIN, COS, LOG,
TAN, EXP, SINH, COSH, TANH, SQRT, LN, PI, MOD,
ROOT, NUM, VAR, ERROR }

public class Token{
     string text;
     int begin;
     int end;
     TokenType type;
     public Token(TokenType type, String text,
int begin, int end)
     {
          this.type = type;
          this.text = text;
          this.begin = begin;
          this.end = end;
     }
     public String toString()
     {
          return String.Format("[{0}, {1}, {2},
{3}]", type, text, begin, end);
     }
}
```

Options:
The region in which the user define scanner class name, the method that returns the next token and it return type, and macros.

For example:

```
%%
%class Lexer
%public
%function getNextToken
%type Token
%line
%char
DIGITS = [0-9]|[1-9][0-9]*
WS = [\r\t\n\f\ ]*
%%
```

Lexer rules:
The region in which defining the regular expression which specifies the reserved words or the chosen tokens.
The rules are used to define the lexical analysis function. Each rule has two parts a regular expression and an action. Each rule starts with `<YYINITIAL>` followed by a regular expression followed by the action.

For example:

```
<YYINITIAL> [a-zA-Z][a-zA-Z0-9]* {return new
Token(TokenType.VAR, yytext(), yychar,
yychar+yytext().Length);}
```

Macros

Macros are a short cut for a long regular expressions which is used multiple times in the regular expression region in the lexer specification file.
For example, instead of writing a regular expression that detects number as: `([0-9]|[1-9][0-9]*)|([0-9]|[1-9][0-9]*)\.([0-9]|[1-9][0-9]*)`

We can define a macro:
`DIGITS = [0-9]|[1-9][0-9]*`

Then define a number as:
`{DIGITS}|{DIGITS}\.{DIGITS}`

This way we can save time and reduce the complexity of the regular expression.

The tokens we used

Token Type	Describtion
ADD	Addition operator "+".
SUB	Subtract operator "-".
DEV	Division operator "/".
MULTI	Multiplying operator "*".
POW	Power symbol "^".
LPARENT	Left parentheses "(".
RPARENT	Right parentheses ")".

EQUAL	Equal sign "=".
SIN, COS, SINH, COSH, TAN, and TANH	the mathematical trigonometry functions.
LOG	Logarithm function.
LN	The natural logarithm.
EXP	The exponential sign "^".
SQRT and ROOT	Square root.
NUM	The set of all numbers VAR : the indefinite variables in the equation e.g. "x or y".
ERROR	Anything that isn't defined above.
EOF	End of file.

Lex file

The lexer file is the file in which the Lexer specification is defined. The Lexer generator provides a full and minimized DFA, It uses regular expressions to describe classes of words. A program fragment is associated with each class of words. This information is given to Lexer as a specification (a Lexer program). Lexer produces a program for a function that can be used to perform lexical analysis. The function finds the longest word starting from the current position in the input stream that is executed by the program fragment associated with the class, and sets the current position in the input stream to be the character after the word. The program fragment has the actual text of the word available to it.

Chapter 3

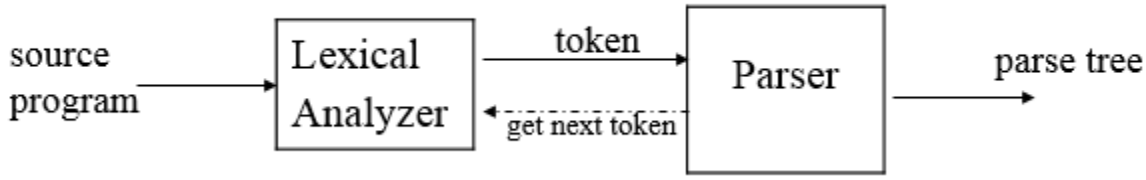
Parsing Analysis

Parsing

Parsing gives the syntactic structure of the given source program as a Parse Tree.

Where lexical analysis splits the input into tokens, the purpose of syntax analysis (also known as parsing) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This something is typically a data structure called the syntax tree of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labelled.

The syntax of a program is described by a Context Free Grammar and Parser checks whether a given source program satisfies the rules implied by a context-free grammar or not. If it satisfies, the parser creates the parse tree of that program. Otherwise, the parser gives the error messages.



Grammar

Grammar is a list of rules which can be used to produce or generate all language's Tokens.

In a context-free grammar, we have:

- A finite set of terminals
- A finite set of non-terminals (syntactic-variables)
- A finite set of productions rules in the following form

- Example:

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid -E$

$E \rightarrow (E)$

$E \rightarrow id$

LL(1) Parsing

- Recursive Descent which grammar structure is directly translated into the structure of a program, this means that each non-terminal is implemented by a function.

  Example:-

```
        Term parse ()
        {
            getNextToken();
            Term t1 = expression();
            if (!eq.IsEquation && lookahead.type
== TokenType.EOF) return t1;

            if (!eq.IsEquation && lookahead.type
!= TokenType.EOF)
```

```
                throw WrongSymbol("End of File",
lookahead.text);

            if (eq.IsEquation && lookahead.type !=
TokenType.EQUAL) throw WrongSymbol("=",
lookahead.text);
            getNextToken();
            Term t2 = expression();
            t1.Substract(t2);
            return t1;
        }
```

Our grammar

For the purpose of this project we created a grammar that we think fits the functionality we want, supporting different types of factors, mathematical function (e.g. sin, cos, tan…) and variables raised to certain powers, equations and expressions.

For the following grammars we defined terminals like NUM, VAR, and FUNCTION. These terminals' definition can be found in the previous chapter where NUM is number, VAR is Identifier and FUNCTION are mathematical functions.

The "value" rule isn't factored for simplification.

equation → expression equation’
equation’ → = expression | ε

expression → signed_term sum_op

sum_op → + term sum_op
sum_op → - term sum_op
sum_op → ε

signed_term → + term
signed_term → - term
signed_term → term

term → factor term_op

term_op → * signed_factor term_op
term_op → / signed_factor term_op
term_op → ε

signed_factor → + factor
signed_factor → - factor
signed_factor → ε

factor → value
factor → (expression)

value → NUM | NUM ^ NUM
value → VAR | VAR ^ NUM
value → FUNCTION NUM | FUNCTION (NUM)

Mathematical Equation Model

To create a mathematical model we used the same structure of the mathematical equation by definition it consists of terms and factors.

Factor:

Factor is a class that defines a mathematical factor, in our code a factor is the smallest part of the mathematical equation for example (2x and 1 in 2x+1=0) and (-3 in y-3).

A factor is divided further to a coefficient and a set of variable and power pairs.

Factor class interface consists of:

- `Public decimal Coefficient`
  returns the coefficient part of the factor.

- `public void AddVariable(string variable, decimal power = 1)`
  Multiplies a variable and adds it to the current variable list making the necessary checks such as deleting the variable if its power changed to 0 after the operation and checking if it already exist adding the powers… e.g. (x * 2x).

- `public void Multiply (Factor f)`
  Multiplies two factors storing the result into the current factor using the following algorithm. e.g. (2x * 3y).

```
multiply coefficients;
if coefficients = 0
     remove the factors
for each variable in the new factor
     multiply the current factor by the variable
```

- `public void Divide (Factor f)`
  Divides two factors storing the result in the current factor, using the following algorithm. e.g. (2x / 3y)

```
divide coefficients;
if coefficients = 0
     remove the factors
for each variable in the new factor
     variable power = power * -1
     multiply the current factor by the variable
```

- `public string VariablesToString()`
  Returns the variables' string representation delimited by ';' e.g. (x^2;y^2).

These were factor's definition and operations.

Term:

A term is a collection of factors added to or subtracted from each other e.g. (2x + y) or (3+z).

The class public interface consists of:

- `public void Add (Factor f)`
  Adds a factor to the current term. eg. 2x + (x+y).

```
for each factor in the current term
     if factor.VariablesToString =
     f.VariablesToString
          add both coefficients
     factor.Add(newFactor)
```

- `public void Subtract (Factor f)`
  Subtracts a factor frome the current term. e.g. (2x+1) -3

```
f.coefficient *= -1
Add(f);
```

- `public void Add (Term t)`
  Adds two terms storing the result in the current term.
  e.g. (2x+1)-(x-3)

```
define a new Term result
for each lFactor in this.Factors and rFactor in
t.Factors:
```

```
        if lFactor.VariablesToString() ==
        rFactor.VariablesToString():
             lFactor.Coefficient +=
             rFactor.Coefficient
             if lFactor.Coefficint != 0:
                  result.Add(lFactor)
             remove lFactor
             remove rFactor
        else:
             result.Add(lFactor)
             result.Add(rFactor)
             remove lFactor
             remove rFactor
   this.Factors = result.Factors
```

- `public void Subtract (Term t)`
  Subtracts two terms and stores the result in the current one.
  e.g. (2x + 1) - (3x - 1)

```
for each factor in t:
     factor.Coefficient *= -1

Add(t)
```

- `public void Multiply (Factor f)`
  Multiplies by a factor storing the result in the current term.
  e.g. 2x * (3y+1)

```
for each factor in Factors:
     factor.Multiply(f)
```

- `public void Divide (Factor f)`
  Divides the current term by a factor. e.g. (3x+1)/2

```
define new Factor temp
for each factor in Factors:
     temp = f
     temp.Divide(f)
     factor = temp
```

- `public void Multiply (Term t)`
  Multiplies two terms and store the result in the current term. e.g. (x+1)(x+2)

```
define new Term result
define new Term rTerm
for each factor in Factors:
     rTerm = t
     rTerm.Multiply(factor)
     result.Add(rTerm)
```

- `public void Divide (Term t)`
  Divides two terms and store the result in the current term. e.g. (2x+1)/(x+3)

```
define new Term result
define new Term rTerm
for each factor in Factors:
     rTerm = t
     rTerm.Divide(factor)
     result.Add(rTerm)
```

Solver

For this project we picked Math.NET mainly because it's in C# and portable to many platforms which fits perfectly our project. Math.NET is described by its creators as:

"Math.NET Numeric aims to provide methods and algorithms for numerical computations in science. Covered topics include special functions, linear algebra, probability models, random numbers, interpolation, integration, regression, optimization problems and more.

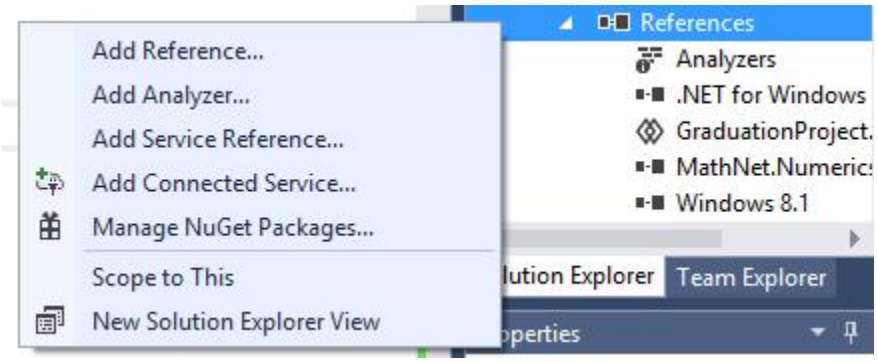
Math.NET numeric is part of the Math.NET initiative and is the result of merging analytics with Math.NET Iridium, replacing both. Available for free under the MIT/X11 License. It targets Microsoft .Net 4, .Net 3.5 and Mono (Windows, Linux and Mac), Silverlight 5, Windows Phone 8 and 8.1, Windows 8/Store (PCL 7, 47, 78, 259 and 328) and Android/iOS (Xamarin). In addition to a purely managed implementation it also supports native hardware optimization."

Math.NET receives a matrix representation and returns a vector with the solution of the system.

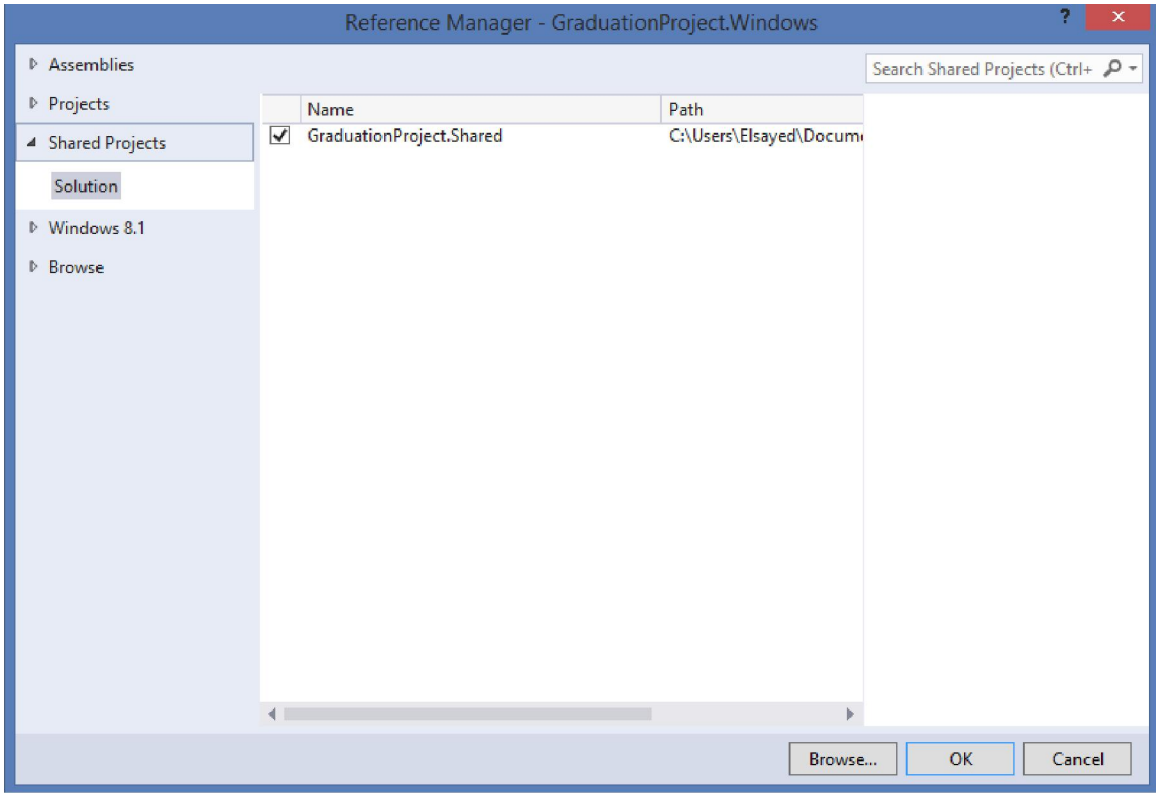
Usage

We produce a c# project that can be reference in any c# project, being cross-platform it can be referenced in solutions targeting desktop, mobile and web.

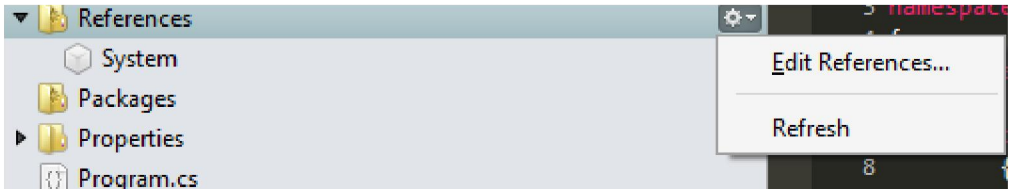
To reference it in Visual Studio Add it to your solution then in your project find references folder, then right click it and choose “Add Reference...”.



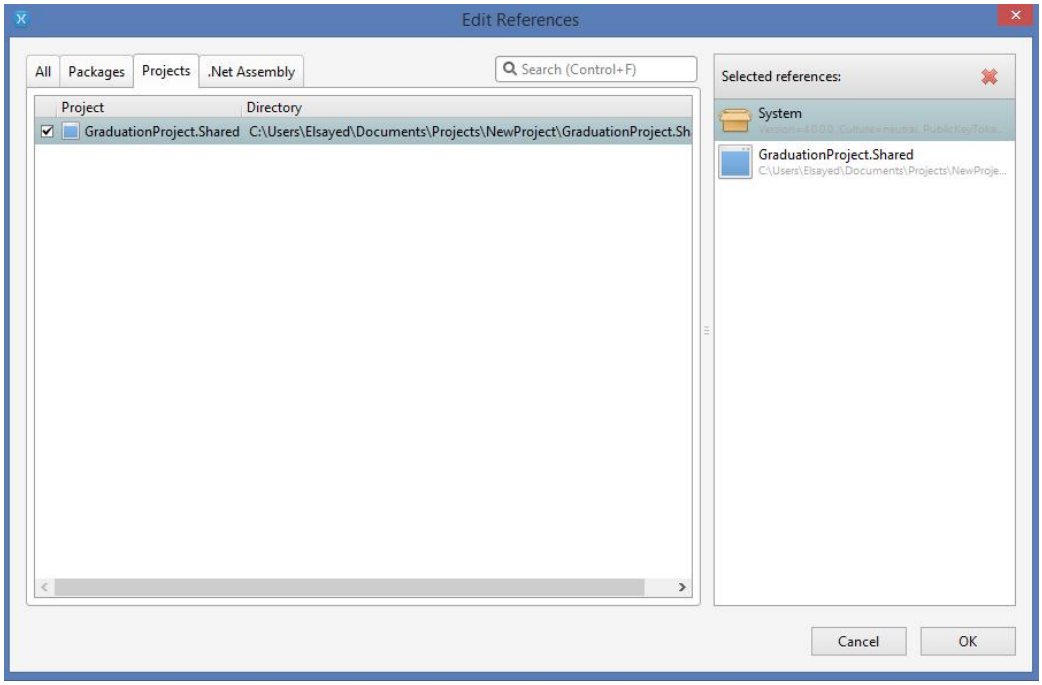
From the “Reference Manager” find the project and mark it then click OK.



In Xamarin Studio and MonoDevelop after adding the project to your solution, find “References” folder, right click on it or click on the gear button then choose “Edit References…”



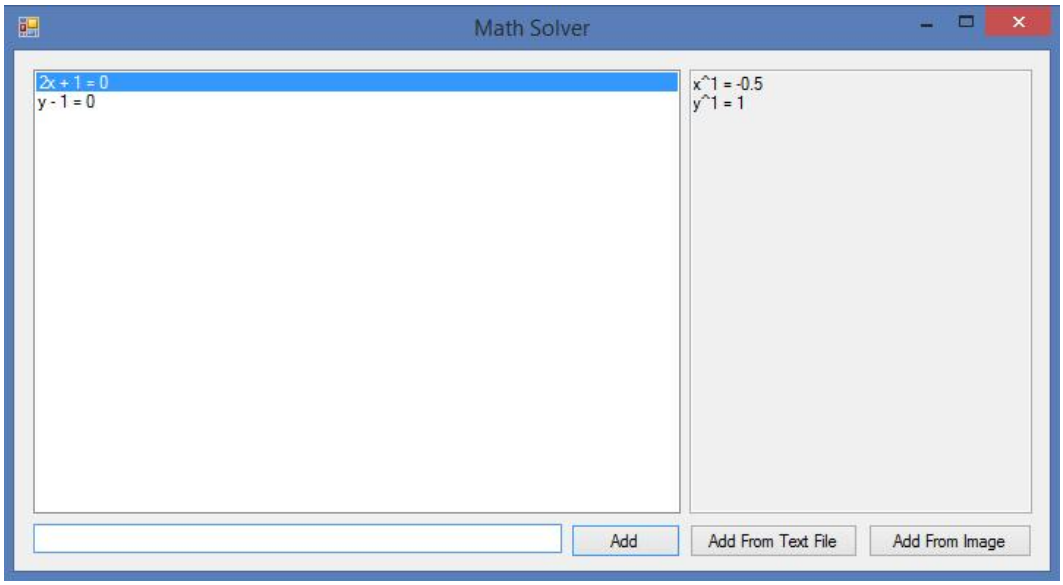
From “Edit References” Window choose Projects then mark our project then click OK.



Then from code you can start using the project to create your own application.

Our Application

We created an application that takes user input in various ways (Text files, Image files and Standard text box).



Our Enhancements

In future, we aim to target more platforms and add more features like:

- Arabic OCR.
- Speech to text.
- Solving more types of equations.