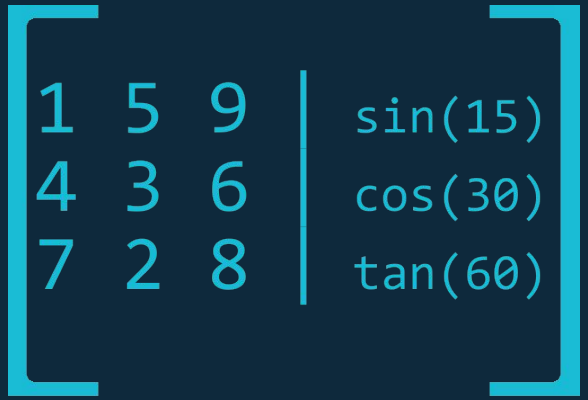
Solving a System of Linear Equations with Matrix Algorithms




```
x = -0.0976
y = -0.1386
z =  0.1601
```

What it does: the basics

- Essentially, it gets three pieces of information from the user and outputs the result. These are:
  - The number of equations,
  - The equations themselves,
  - And the variables used.

- From this, it prints out the solution to the system of equations provided.

```
How many equations would you like?
2
Please enter equation number 1.
x - 2y = 3
Please enter equation number 2.
2x - y = -3
What variables did you use? Please enter them with the syntax of x, y, z.
x, y
x = -3
y = -3
```

The user-shown parts of the code.

How it was implemented

- The program was coded in the language of C++ through the use of object-oriented programming, a style that makes use of user-created ‘classes’ in its operation.

- Collaboration was facilitated by the software known as Git (more specifically, the website of Github), a useful tool enabling multiple people to contribute independently to the code without inconveniencing each other.

How it works: a step-by-step analysis

The main method

- In coding, the main method is what the computer actually runs when you call it with the executable.
- In our case, the main method is very simple: an infinite loop of system creation that repeats until the user enters ‘exit.’
- Each iteration of the loop creates a new `SystemsOfEquations` object, and it is in here that the main part of the code is run.

The constructor method

- In object-oriented programming, a constructor is what builds the object. When it gets called, it takes certain parameters as inputs and creates the object based off of those.
- In our case, the constructor gets its inputs directly entered by the user from the console, eliminating the need for parameters.
- The constructor for the `SystemsOfEquations` object is composed of four main sections:
  - Initialisation of variables,
  - Equation entry,
  - Equation parsing,
  - And solution calculating.
- (There's also a part which calls the help method, a simple few lines of code that print out the guidelines for inputs. They **must** follow the specifications given.)

Initialisation of variables

- The constructor initialises several empty variables, such as the integer storing the number of equations, the string vector storing the raw equations, the character vector storing the variables used, and many more.

```
this->equations = {};
this->eqsUnparsed = {};
this->singEqUnparsed = {};
this->varsUsed = {};
this->mat = {};
this->matFunc = {};
this->matNum = {};
this->post = {};
this->postFunc = {};
this->postEqs = {};
this->augMat = {};
this->solns = {};
```

Don't worry about interpreting the names — the important part is that they now exist in memory.

Equation entry

- The constructor first prompts the user for the number of equations they wish to enter. This is stored as `numOfEqs`.
- From here, the constructor then loops as many times as there are equations to get the equations themselves from the user. As they are inputted, they get pushed to a string vector (a vector is an object type that stores a list of other objects).
- The last part of equation entry — and the last thing the user directly sees before the solutions are printed — is variable entry. The user inputs a comma-separated series of characters, which are then themselves pushed to the back of a character vector.

Equation parsing, part 1

- The equation parsing method is where the bulk of the code's complexity lies.
- It iterates through the string vector of raw equations, and performs various operations on each equation to acquire useful information from it.
- Let an example equation be 'x - 2y = -sin(30).'
- The first operation it performs is a removing of spaces and replacing of '-'s with '+-' (unless they're before an equals sign). This is necessary because, later, it splits up the equation by + signs.
  - Equation at this point: 'x+-2y=-sin(30)'
- It then finds coefficient-less variables and inserts a 1 before them.
  - Equation at this point: '1x+-2y=-sin(30)'

Equation parsing, part 2

- At this point, it splits the equation into two parts — the part before the equals sign and the part after it. (1x+-2y // -sin(30))
- Let's look at how it parses the pre-equals sign part of the equation first (though it pretty much does the exact same thing for the post-equals sign part as well, just without the variable bit)
- It iterates through the equation and finds where there are variables. Then, it adds every character from the last + to the variable to the corresponding element of the raw matrix.
    - At present, it looks something like this:

1	-2
--	--

Equation parsing, part 3

- Once it's got all that, it looks through the elements of the matrix to put spaces after negative signs and replace parentheses (for functions like sin & cos) with spaces.
- By this point, the parsing is almost done — but everything still needs to be turned into numbers.
- This is done by copying the string matrix and then setting every corresponding element to a double, evaluated with some code that takes a string and evaluates it. (Its workings are themselves highly complex, and aren't too important, so they'll be skipped over.)
- Having done this to both the left and right sides of the equals sign, the program then combines the two into the augmented matrix, which is the final step of the parsing.

Evaluation with Gauss-Jordan elimination

Gauss-Jordan elimination

- The algorithm for Gauss-Jordan elimination is fairly simple: a couple for loops and an if statement to make it into diagonal form. It looks like this.

```
double temp = 0;
for(int i = 0; i < matNum.size(); i++){
    for(int j = 0; j < matNum.size(); j++){
        if(i!=j){
            temp = augMat.at(j).at(i)/augMat.at(i).at(i);
            for(int k = 0; k < matNum.size()+1; k++){
                augMat.at(j).at(k) = augMat.at(j).at(k) - temp*augMat.at(i).at(k);
            }
        }
    }
}
```

- `matNum` is the coefficient matrix and `augMat` is the augmented matrix.
- The final step from here is to divide each row by the value in the diagonal — and, with this, it is done!

Miscellaneous information about the program

- One of the most crucial parts of making this work was a debug file — essentially, every time we wrote code to do something, it'd write what it was that was done.

```
debug << "\nSearching for coefficient-less variables." << endl;
for(int i = 0; i < toAdd.size(); i++){
    for(int j = 0; j < varsUsed.size(); j++){
        if(toAdd.at(i)==varsUsed.at(j)){
            debug << "Variable " << toAdd.at(i) << " found at index " << i << "." << endl;
            if(i==0){
                debug << "Entered i is zero if statement." << endl;
                toAdd.insert(i,"1");
                i++;
                debug << "Variable was found to be coefficient-less, so 1 has been supplied." << endl;
            }else if((i!=0)&&((toAdd.at(i-1)=='+'&&toAdd.at(i)!='-')||toAdd.at(i-1)=='-')||toAdd.at(i-1)=='='){
                debug << "Entered i is not zero if statement." << endl;
                toAdd.insert(i,"1");
                i++;
                debug << "Variable was found to be coefficient-less, so 1 has been supplied." << endl;
            }else{
                debug << "Has coefficient, and so nothing shall be done." << endl;
            }
        }
    }
}
```

Miscellaneous information about the program

A little snippet of some of the debug output

```
Removing variables from matrix.
Variable at position 0,0 x removed.
Variable at position 0,1 y removed.
Variable at position 1,0 x removed.
Variable at position 1,1 y removed.
Matrix without variables:
1 -2
2 -1
Coefficient matrix:
3
-3
Checking for functions.
Checking for negative signs.
Negative sign found at position 0,1, index 0; inserting space after.
Negative sign found at position 1,1, index 0; inserting space after.
Checking for functions in the coefficient matrix.
Checking for negative signs in the coefficient matrix.
Negative sign found at position 1,0, index 0; inserting space after.
Current matrix:
1 - 2
2 - 1
Current coefficient matrix:
3
- 3
Converting string matrix into a double matrix.
Converting coefficient string matrix into a double matrix.
Dimensions of postEqs matrix: 2x1
Dimensions of postFunc matrix: 2x1
Matrix converted into doubles:
1 -2
2 -1
Coefficient matrix converted into doubles:
3
-3
Beginning Gauss-Jordan elimination.
Combining matrices into an augmented matrix.
Augmented matrix:
1 -2 3
2 -1 -3
Matrix in diagonal form:
1 0 -3
0 3 -9
Calculating solutions from matrix.
Solutions:
x = -3
y = -3
Printing solutions.
```

Bibliography

- Jude