[](http://www.google.co.uk/url?sa=i&rct=j&q=pates+grammar&source=images&cd=&cad=rja&uact=8&ved=&url=http://www.schooltogs.com/schooltogs&ei=JaI3VYXJBYrhaIyNgYAC&psig=AFQjCNFVEgg6wkOcJvB9BV_eWuOJO_YOEg&ust=1429795749580256)



**PATE’S GRAMMAR SCHOOL**

**COMPUTING DEPARTMENT**

**Unit 3/4 – Programming Project**

CANDIDATE NAME

Alex Daniels

EXAM NUMBER

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[(1) Anlysis of the problem (10 Marks) 3](#_Toc454442805)

[(i) problem definition 3](#_Toc454442806)

[(ii) Stakeholders 5](#_Toc454442807)

[(iii) RESEARCH THE PROBLEM 6](#_Toc454442808)

[(iv) SPECIFIY THE PROPOSED SOLUTION 9](#_Toc454442809)

[(2) Design OF THE SOLUTION [15 marks] 11](#_Toc454442810)

[(i) DECOMPOSE THE PROBLEM 11](#_Toc454442811)

[(ii) DESCRIBE THE SOLUTION 12](#_Toc454442812)

[(iii) DESCRIBE THE APPROACH TO TESTING 13](#_Toc454442813)

[(3) DEVELOPING THE SOLUTION (25 mARKS) 14](#_Toc454442814)

[(i) INTERATIVE DEVELOPMENT PROCESS 14](#_Toc454442815)

[(ii) TESTING TO INFORM DEVELOPMENT 15](#_Toc454442816)

[(4) EVALUATION (20 mARKS) 16](#_Toc454442817)

[(I) TESTING TO INFORM EVALUATION 16](#_Toc454442818)

[(II) SUCCESS OF THE SOLUTION 17](#_Toc454442819)

[(III) DESCRIBE THE FINAL PRODUCT 18](#_Toc454442820)

[(IV) MAINTENANCE AND DEVELOPMENT 19](#_Toc454442821)

[APPENDIX A - BIBLIOGRAPHY 20](#_Toc454442822)

# (1) Anlysis of the problem (10 Marks)

## (i) problem definition

*(a) Describe and justify the features that make the problem solvable by computational methods.*

*(b) Explain why the problem is amenable to a computational approach.*

Invented by German engineer Arthur Scherbius in 1918, the Enigma machine is cipher based device designed to encrypt messages, unable to be read by anyone but the intended recipient with a symmetric key to the message. It was developed and used extensively by the Nazi Government throughout the 30’s, and going into WW2 was one of their most powerful weapons. It allowed German soldiers the capability to send messages over the new radio and morse code quickly and, most importantly, securely. This is why solving the Enigma cipher became a top priority for the Allies, culminating in Alan Turing’s work at Bletchley park, where he invented the Bombe attack in 1940 and consequently greatly advanced the fields of cryptography and computational science. 80 years later, I want to create an educational program to first explain and simulate an Enigma machine so that it can be understood why the machine was so difficult to solve, and in the same program it should be able to solve an Enigma cipher given only ciphertext.

The original Enigma ciphers contained no plugboards, so we shall start here. The process of encrypting a message begins by a button being pressed on the Enigma machines keyboard. This rotates the rotors of the machine according to a pattern, with the right rotor moving every key press, the middle every 26th key press (with double steps) and the leftmost rotor stepping every 26th movement of the middle rotor (or 25th permutation due to a quirk of the enigma machines, double stepping). Electrical current is then passed through the key into the rightmost rotor, which consists of a set of wires essentially substituting the letters. This passes into the next rotor and is repeated until hitting the end, where the electrical current hits a reflector and is substituted yet again, and sent backwards through the rotors (this fact leads to no letter encrypting to itself, a very useful property for code breakers). In later, more complex machines, a plugboard was introduced which would swap two letters at both the input and output, increasing the settings space considerably. Later machines would even have a 4th stationary rotor, and increased variety of rotors. Overall, Bletchley park would need to know: which rotors are in use and at which position at the start of encryption, what offset the rotors had from their stepping position, and the plugboard settings.

Given this, simulating an Enigma machine is a task very well suited to computational methods, given the complicated and involved process that goes into encrypting a letter. A typical encryption requires keeping track of 10 letters, and going through 9 lookups, all while advancing the state of the machine and checking for rotor position changes, and while on paper it can be managed for a few letters, this becomes out of hand very quickly. This is why the original cipher made use of the relatively modern electromechanical system, and a computer can handle this series of operations just as well, being able to store the state of a system very easily and being able to look up substitutions very quickly through array-like structures. Furthermore, an electromechanical system such as was originally used would not be well suited to the task of education due to one simple reason: electricity isn’t easy to see. It becomes very unfeasible very fast to show the internal state of the enigma machine during an encryption due to the sheer number of lights that would be needed – and where they would need to be placed. This is easily solved with a computer, being able to lay out and display the state of the machine without regard to physical dimensions.

Finally, solving an Enigma cipher is incredibly well suited to computational approaches. Due to the straightforward process of ciphering a string, computers can calculate millions of potential keys through iteration, all in a relatively short period. This is far faster than working on paper, with the most effective non-electronic methods for solving Enigma ciphers involving complex plastic sheets and light panels that scaled completely ineffectively, only being practical during the 1930’s when Enigma machines were still only using 3 rotors. Even then, Turing’s Bombe attack was limited in scope and speed, requiring complicated mathematical menus be constructed for each decryption and multiple false starts were produced on the bombe machines per run. In the end, the Bombe attack took somewhere between half an hour to 2 hours per key to decrypt, severely limiting the allies’ responsiveness. In comparison, this technique on modern computers may take somewhere between a minute to a few seconds, orders of magnitude faster.

## (ii) Stakeholders

1. *Identify and describe those who will have an interest in the solution explaining how the solution is appropriate to their needs (this may be named individuals, groups or persona that describes the target end user).*

My program will be designed to appeal to anyone interested in the history of cryptography, but more specifically I will target 16-25 year-olds with a background in maths, linguistics or computer science.

I have 3 specific stakeholders in mind:

Ben Carter – 18 years old, mathematics and computer science student, with backgrounds in cybersecurity. Ben is relevant to this project as they have interests in the history of computers and cybersecurity, wanting to understand the relevance of Enigma to this subject through an understanding of the underlying algorithm and machine.

Sam McWhirter – 18 years old, computer science student, with interest in WW2 history and related Enigma relevance. Sam is relevant to this project as he has had an interest in how the Enigma machine turned the tide of WW2, wishing to see why it was such a difficult problem to decrypt and recognizing the value that the team at Bletchley Park added to the allies in their solving of it.

Henry Warburton – 17 years old, maths and linguistics student, interested in cryptography and has made many hobbyist ciphers and cryptographic systems for fun. He is relevant to this project as he has lots of experience with complex and interesting cryptography throughout history and wants to add yet another cipher to his library of knowledge. However, the currently established websites for this purpose are just not interactive enough for him, requiring following along with long ciphers by paper and pen, not allowing quick feedback when he makes mistakes.

I then asked each of them a questionnaire.

|  |  |  |  |
| --- | --- | --- | --- |
| Question | Ben | Sam | Henry |
| How well do you already understand the engima cipher | I know of the individual components of the machine, but I’m not entirely sure of how the cipher and whole process works | Passing knowledge and understanding. | I understand that it was used in ww2 by the Germans to send secret messages and it was eventually cracked at Bletchley park, however I don’t know how it works |
| Do you understand many Cryptographic ciphers | Yes – simple ciphers such as substitution, one-time pad as well as modern computer ciphers e.g RSA, AES DES | Yes. I understand some simple ciphers, with my most complex being Vigenere cipher | I understand a few rudimentary encoding methods like pig pen and rail fence and caesar ciphers and code wheels and grilles and dot code |
| Could you perform any ciphers on paper | In theory – yes, but modern computer ciphers could take days | Yes, the Caesar cipher. | Probably all of the ones I just mentioned however they’re not very secure so I could also crack them on paper |
| How did you learn about any of these ciphers | Online tools such as cyberchef, a cybersecurity programme called cyberdiscovery | I know this because of “Gravity Falls”, where it was integral part of the lore. | My favourite book when I was younger called the Knowhow Omnibus and it had a section on spying which had lots of information about codes and ciphers |
| Why would you like to learn about the enigma cipher | I enjoy looking at how modern computers can solve a range of old problems, and I am impressed by how these problems were solved in their time. | I love WW2, and I want to learn more about the technology used to destroy the Nazis. | I like cryptology and it seems interesting |
| Are there any features you would like in an enigma cipher educational tool | I’d like to see how data travels through the Enigma. I’d also like to see an estimation of how long it would take for a bombe machine to solve a given cipher. | Text-to-Speech; I am dyslexic | I would like it to show how the physical machine worked and the history of what went into solving how to break it |

As can be seen, the questionnaire shows that despite these people all coming from quite cryptographic/ computer science based areas, none of them have a particularly extensive knowledge of the Enigma machine. Furthermore, they would all be interested in learning them, given the lack of tools online for learning a lot of cryptography, resorting to more technical programs or books to understand the programs, a less refined method of teaching. Finally, I will be sure to include some history of the Bombe attack in my program someway, and some accessibility capabilities should be introduced to the program.

## (iii) RESEARCH THE PROBLEM

*(a) Research the problem and solutions to similar problems to identify and justify suitable approaches to a solution.*

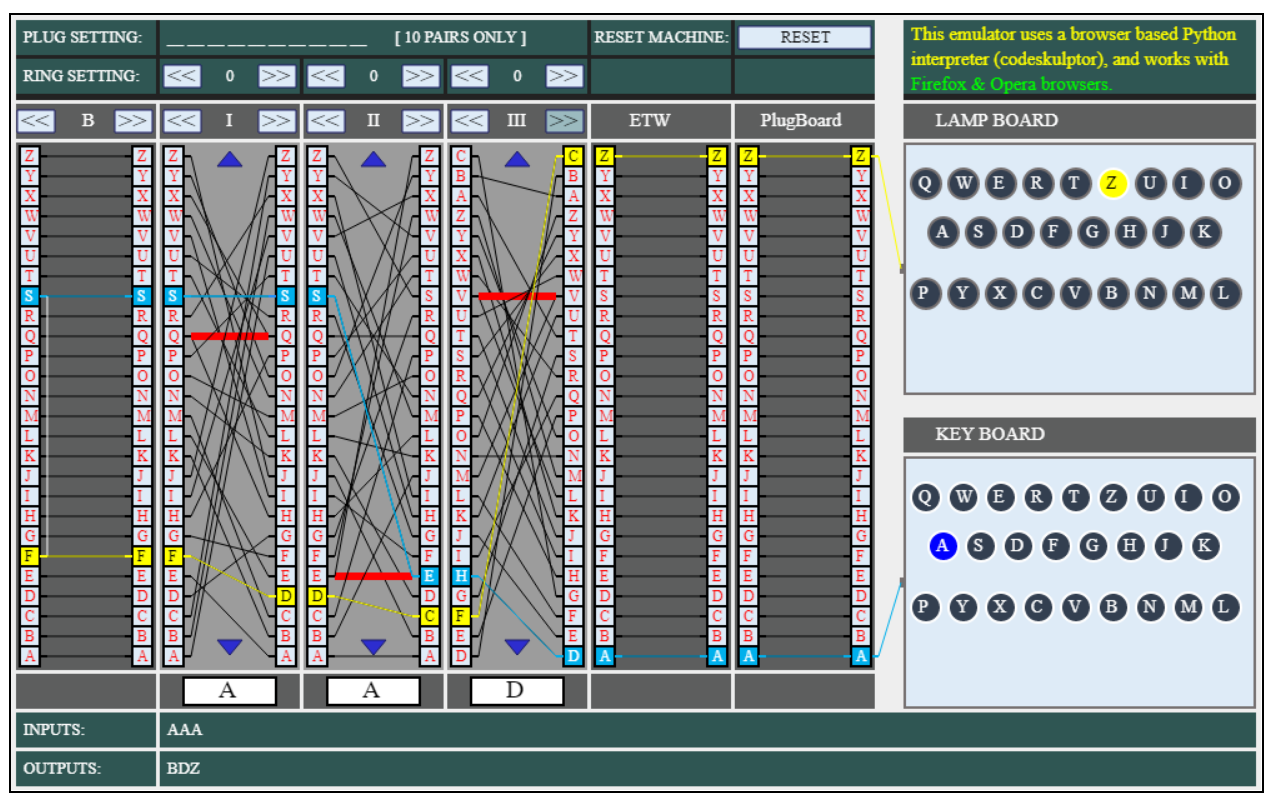
*(b) Describe the essential features of a computational solution explaining these choices.*

*(c) Explain the limitations of the proposed solution.*

<https://piotte13.github.io/enigma-cipher/>

Graphical user interface, diagram

Description automatically generated



*The website encrypting AAA with rotors I II III in position AAA and ring settings 111. As you can see the rotors have mutated and the website uses offset ring settings.*

This website simulates an Enigma machine in your browser. Made in python and interpreted, this site shows the internal workings of the Enigma machine and all it’s functional parts in a visual way. The user can use any of the buttons provided to change a setting such as move/change a rotor or press a key, and can click on 2 plugboard buttons to create plug connection between them. When a key is pressed, that input on the plugboard will light up in blue and each subsequent line through the encryption will light up, changing colour at the reflector to yellow. Each rotor connection is displayed, and rotate in real time in line with the settings provided.  
What I liked, and will try to implement:

* The abstract layout and wire representations
* The colour scheme, with the less vibrant colours being easy on the eyes and professional
* The ability to step through the process and understand what is going on through lit up wires
* The settings of the machine can be manually changed at any point, allowing the user to learn much faster through interaction
* Is an easy to access and use web application, requiring no interpreter or downloads. However I am uncomfortable in my knowledge of web development and don’t think a website can do the heavy lifting required for solving Enigma ciphers, so I am limited to a desktop app

What I disliked about the implementation and will try to change:

* Some settings do not line up with the well-recognized format for enigma settings, eg ring settings are 0 indexed and not 1 indexed like most implementations
* The input system for the settings, while interactive, is quite slow, and I should like the ring/rotor position settings to stay when changing rotor
* Lack of variety of Enigma machine types (only simulating an M3 model)
* Entering a plugboard configuration is very slow and could be easily sped up with a text box
* No solver feature

[https://web.archive.org/web/20060720040135/http://members.fortunecity.com/jpeschel/gillog1.htm](https://web.archive.org/web/20060720040135/http:/members.fortunecity.com/jpeschel/gillog1.htm)

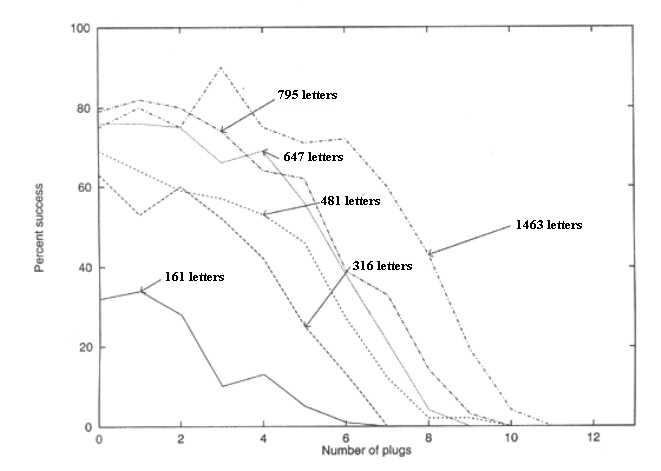
Text

Description automatically generated

A picture containing text

Description automatically generated

*Index of co-incidence formula as given by paper*



*Diagram from paper showing the effectiveness of the algorithm based on length of input and number of plugs. Effectiveness calculated as percentage of inputs deciphered to 80% similarity with plaintext*

This paper from 1995 details an algorithm for solving the Enigma cipher with modern computational techniques, and tests this algorithm on various WW2 ciphers.

What I liked and will implement:

* The stages of the algorithm used, ie solve rotors, solve rings, solve plugboard
* Testing on a wide range of inputs and settings to benchmark the solver
* The fitness functions used, especially earlier on i.e the Index of Coincidence test
* Being able to see the stages of decryption

What I will change in my implementation:

* Have options to repeat certain parts of the algorithm with other known settings, i.e re-solve the rotors after finding the plugboard configuration
* Use a stepped function test for the last stages of the decryption/plugboard hill climb. So instead of using trigrams or bigrams for all 10 plugs, the first 4 could use bigrams, next 4 could use trigrams and the final 2 could use quadgrams, reflecting the effectiveness as more plugs are introduced
* Multithreading could speed up the rotor position finder considerably
* Due to the lack of publicly available historical Enigma messages available, I will not use any for my evaluation and n-gram building like the author of this paper does. Instead, I will use English prose, as that is what will most likely to be given as input.

<https://github.com/mikepound/enigma>

Graphical user interface

Description automatically generated with medium confidence

*A sample deciphering. As can be seen, the correct decryption was the 2nd best rotor configuration. As can be also seen, this was entirely done with the command prompt.*

This is a Java implementation of the previously mentioned paper, made as a demonstration for a computerphile youtube video. The main branch of the code is unoptimized to cater to the youtube audience, but has a more optimized branch making use of multithreading.

What I liked:

* Stages of the decryption process can be seen and understood, with decryption and fitness updates at every stage of the process
* The optimized branch making use of multithreading is very fast, with it only taking a few seconds to fully decrypt a message on my machine – however this may be due to the usage of java, which my design in python may not be able to keep up with
* The code is very modular and object oriented, allowing sections of the algorithm to be modified and mixed and matched very easily
* The usage of numbers instead of letters inside the code allows fast substitutions to be made and math to be performed on the characters without the overhead of switching to and from chars

What I disliked:

* No GUI, or any kind of user interface. All interaction with the program is through java source files
* The algorithm could be improved somewhat, only taking the top result from the rotor selection to the next stage, which statistically is unlikely to be the original rotor settings
* Being written in java, I am unable to use any of the libraries to test my code
* While being very vocal about the stages of the decryption, the program provides somewhat too much feedback during the rotor finding stage, printing to the screen every time a rotor is searched

Limitations

I will develop my program for a python interpreter to run, as it is the language I am most comfortable with and has good support for all areas of the project. However this will lead to some limitations. Namely, I will be unable to produce a high quality, modern GUI and I will be unable to reach particularly fast decryptions, especially on larger Enigma machine variants (e.g the M4) however, to combat this I will write the solver section of the program in C/C++ and call this code to leverage the performance of C code. I may also leverage some web development tools through python libraries to make my interface look a lot better.

Computational solution

Abstraction and visualisation

The key objects will be the machine settings, the machine being emulated and the solver. The rotors can be represented abstractly as lines between letters on either side of the rotor, being moved up and down to represent turning. The plugboard and reflector can represented similarly, but statically.

Thinking ahead

Inputs will be mostly through the mouse and clicking, but some may be done through the keyboard for inputting phrases or settings.

Output will be structured over 3 tabs, one for Enigma emulation, one for specifying the details of the Enigma being emulated and one for solving an Enigma cipher.

Thinking procedurally

The program will be based around the function encrypt(), and will take in a series of settings and a letter and encrypt a letter, while mutating the settings. Multiple encrypt() calls together will construct a messageEncrypt(), and lots of messageEncrypt() with different settings will make up the bulk of the solver algorithm, with stages for first iterating over the rotors (findRotorSettings()), then the rings (findRingSettings()), then the plugboard (findPlugs()). All these problems building on one another makes the problem very amenable to a computational approach, with changing one base procedure propagating up to even the highest level procedures with fixes. Furthermore the messageEncrypt() can be used in the main emulator given that the program stores the state. User interaction will be handled by a various functions being called, but a main updating function will be called in a loop to update the GUI should other functions modify it.

Thinking logically

The critical loop statement for the program will be monitoring for user input on the screen, and updating the interface accordingly. The most critical if-statements will be present in the encrypt() procedure, determining whether to shift the middle and left rotors or, in the internal forms, increase the offset by one.

Thinking concurrently

The main thread will be occupied by updating the screen and handling user input, and can handle encryption so quickly as to seem concurrently, however the solver and file loading may require the main thread create multiple threads to handle the more computationally expensive processes while not impacting performance and responsiveness of the GUI.

## (iv) SPECIFIY THE PROPOSED SOLUTION

1. Aesthetic criteria
   1. The design consists of a default window, with variable width but 600 pixels height and non-resizable by the user
   2. The window should have 3 tabs, a simulator tab, a machine details tab and a solver tab. The tabs should be selectable from a list at top of window
   3. Each tab will follow the same colour scheme of green and white
   4. Text input boxes will follow default windows style, with white background and black text with the effect of seeming indented
   5. Action buttons will follow a similar style, white background and black text but an effect of being raised
   6. Simulator tab
      1. Will have a representation of internal electrical wirings on top left of screen
      2. Colours will consist of green background, with white patches indicating a part of the system and black lines symbolizing electrical wiring. Wires will turn yellow when active
      3. Machine state can be input on right of screen via input text boxes
      4. A string can be put in text box on bottom of screen, and the output is given through another text box below that.
      5. Representation can also be used for input output
      6. Variable number of rotors means a variable width must be given to the tab
      7. Letters will be shown between each stage of the machine so user can follow along
   7. Machine detail tab
      1. Green background, with white boxes for input and output
      2. A box with selectable buttons on left, allowing user to select a machine state
      3. Text box below allows user to give their own filepaths to add to box
      4. Rotors (and reflector) of the machine are display on right with text input, allowing user to add or remove rotors and customize them
   8. Solver tab
      1. Will have a text input box top left for ciphertext
      2. Action buttons below so user can specify which operations to attempt
      3. Text boxes with state below buttons will be filled in in throughout solving
      4. Large simple output terminal showing the state of the solver
2. Input criteria
   1. Tabs can be navigated through and selected through using tab buttons at top
   2. Simulator tab
      1. Buttons above each rotor can rotate through selected rotors
      2. Buttons above and below each rotor allow rotation through rotor positions
      3. Buttons above each rotor can rotate through ring settings
      4. Plugboard selection on right by clicking a letter then clicking another, creating a plug setting
      5. Text boxes for all previously mentioned machine settings
      6. User can use their keyboard to enter a letter and watch it be encrypted
      7. Alternatively, a text box can be used to encrypt a full string
   3. Machine Detail Tab
      1. Saved machine details can be selected by clicking
      2. New machine details files can be added by text box with path to file
      3. Machine details can be saved to file through button
      4. Buttons to add/delete rotors from current details
      5. Reflector will be described by text box
      6. Each rotor described by text box for encryption, smaller text box for stepping positions, and flags to describe which positions the rotor can be in
   4. Solver tab
      1. String input given by large (paragraph size) text box
      2. Grid of buttons allow different operations to be performed on ciphertext
      3. Text box inputs for known machine state inputs for particular operations
3. Output criteria
   1. Each tab will contain a box for outputting feedback in the form of error codes and hints
   2. Simulator tab
      1. A text box row below rotors will output recently encrypted letters and encrypted string
      2. Roman numeral numbers in a box will output the current rotor selection above each rotor
      3. Box with letter above each rotor will display current rotor position
      4. Box with number above each rotor will display current ring position
      5. Each rotor/reflector/plugboard will display all it’s connections, with active ones being lit up (not during full text encryption)
   3. Solver tab
      1. There will be a progress box providing the status of the cracking side
      2. Machine settings will be given by text boxes allowing input and output
      3. The final decrypted message will be given
      4. A box providing console-like output will provide data about each step of the decrypting process, such as fitness scores.
4. Processing criteria
   1. Machine specifications will be loaded from files and validated
   2. All inputs will be validated
      1. Plugboard lengths must be even
      2. Rotor/reflector strings must be 26 letters long
      3. Only 3 or 4 rotors can be given
      4. Rotor selection must be less than the number of available rotors, and greater than 0
      5. Ring/rotor position must be between 1 and 26 inclusive
      6. Only letters can be encrypted – numbers and punctuation ignored
   3. Letters will be encrypted via following steps
      1. Mutate rotor positions, according to stepping positions and double stepping
      2. Encrypt through plugboard
      3. Encrypt through rotors, reflector, rotors
      4. Encrypt back through plugboard
   4. Solver will proceed as follows
      1. Check every combination of rotors and rotor positions, find highest 3 scoring according to IoC fitness function
      2. Check every combination of 2nd and 3rd ring positions on the 3 combinations, keeping top 2 by IoC fitness function
      3. Use a hill climbing technique on both remaining rotor/ring positions to find the optimum plugboard configuration. Output highest scoring settings layout according to fitness functions below
      4. Hill climb according to bi-grams from 1-4 plugs, tri-grams from 5-8 plugs, quad-grams from 9-10 plugs.
   5. Solver must “succeed” 50% of the time with an input of length 500, and complete within 20s with 5 rotors available, 1 minute on 8 rotors available, 2 minutes on 8 rotors available with 4 rotor machine. Tested on my 6-core 3.2ghz pc.
5. Platform
   1. The program must run on a windows 10 64-bit computer, as these are the computers I have access to
   2. The program will require python 3.11+
   3. A mouse will be necessary, and a keyboard recommended
   4. //Here I may need to include the requirments of pybind11, such as cmake, visual studio and the lib itself. I don’t know if this is necessary yet.

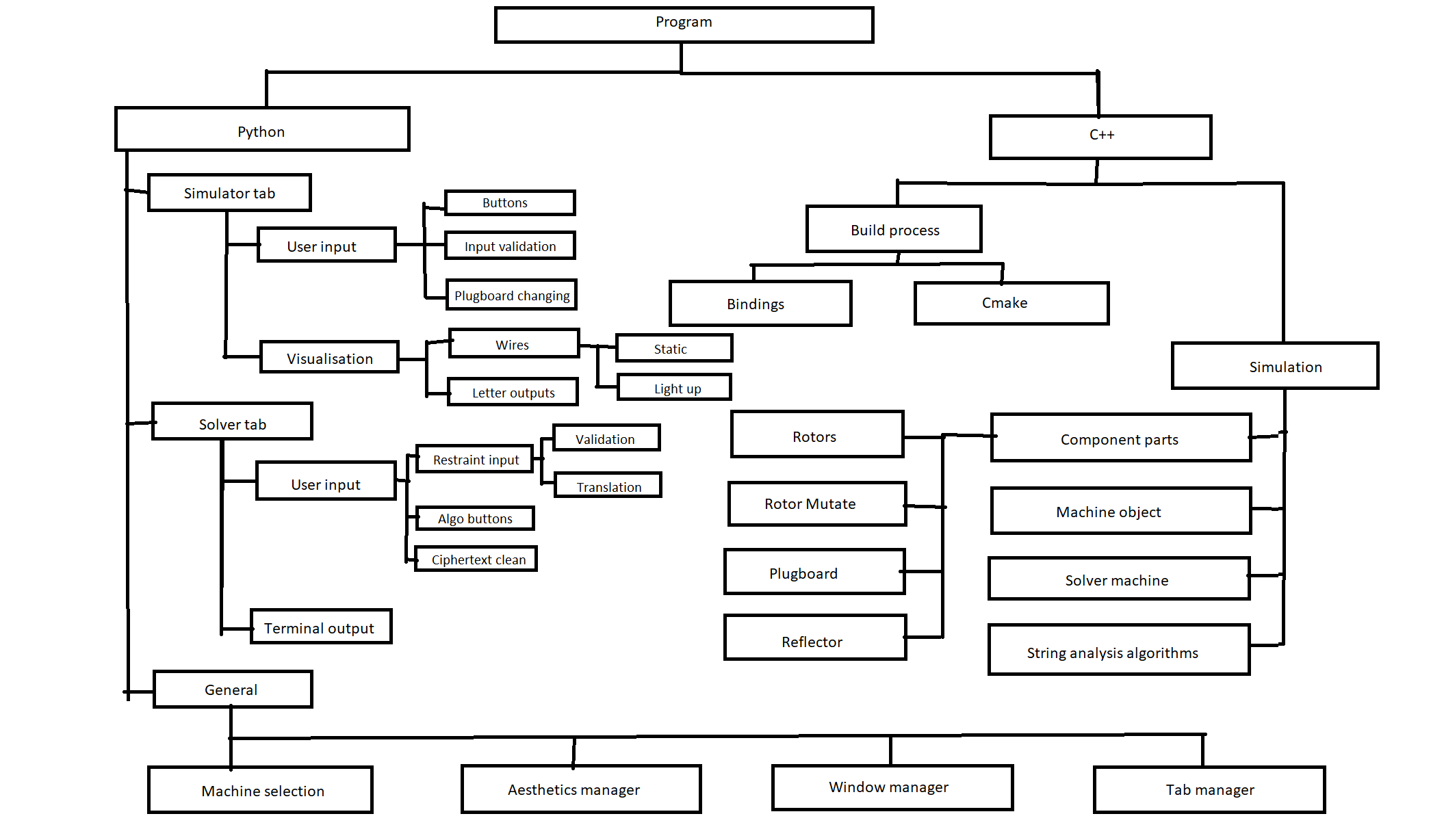
# ``(2) Design OF THE SOLUTION [15 marks]

## (i) DECOMPOSE THE PROBLEM

*(a) Break down the problem into smaller parts suitable for computational solutions justifying any decisions made.*

I have chosen to use a mixture of languages. Python will be used for the bulk of the program, allowing me to leverage my strong understanding and fast development times in python to build the GUI and most data handling. However, a large portion of the program will be coded in C++, using a library called “Pybind11” to generate python bindings. This allows me to write performant code in C++, compile it into a DLL and run in the main python program, satisfying the need for very fast code for the brute forcing steps of solving. C++ and python will also be useful for the machine simulation given their hybrid OOP-procedural style.

However, a big separation from the specification must be made here, in the form of abandoning custom machine specifications. This is to ensure the project can be completed in time, as allowing custom specifications outside of the main I, M3 and M4 will over complicate the design, requiring edge cases and customization to be handled in critical areas.



Here is a top-down view of all the modules and parts for this system to work. On the right can be found the functional C++ components and on the left is the python user interface components. This separation allows abstraction, as when creating the python interface all need be done is call a function on an object and the message will be encrypted without having to consider the consequences and process underlying this.

Diagram

Description automatically generated

This is the inheritance/bottom up diagram of how the machine parts will be simulated and ran. Unincluded is the analysis section, coming under float(\*fitness)(int[]), which was originally a function pointer, however during development this will become a class in its own right designated for evaluating possible solutions.

There will be a clear difference between classes designed to simulate an enigma machine with 3 rotors (e.g the Enigma I and Enigma M3) and those designed to simulate an enigma machine with 4 rotors, however the 4 based classes will inherit from the 3 based classes. This leads to a multiple inheritance in SolverMachine4, inheriting both form the 3 based Solver Machine and the 4 based Machine4. This allows me to reuse plugboard and ring setting finding processes used to solve a 3 rotor enigma machine (as the presence of a 4th rotor will not be relevant here) but used the overridden encryption algorithm of the 4 rotor machine while overriding the rotor finding method to handle 4 rotors.

## (ii) DESCRIBE THE SOLUTION

*(a) Explain and justify the structure of the solution.*

*(b) Describe the parts of the solution using algorithms justifying how these algorithms form a complete solution to the problem.*

*(c) Describe usability features to be included in the solution.*

*(d) Identify key variables / data structures / classes justifying choices and any necessary validation.*

*(e) Identified and justified the test data to be used during the iterative development of the solution.*

### Structure

The structure of the program will be simple. Python and Tkinter will handle the GUI using a main loop called at the end of the main python script, waiting for input from the user. Upon entering any details in the interface and pressing a button, an according function will be called. If needed, these details will be validated and sent to an underlying C++ class to evaluate any changes or encryptions. The entire state of the underlying class will then be read by python and visualised for the user. For example, the send to simulator button on the simulator page will proceed through the following pseudocode.

Func sendToSimulatorPressed

Details = Interface.get\_details

For (rotor in Details.rotors)

If not (rotor in machineSpecification)

ErrorReadout.write(“unknown rotor selected”)

Rotor = machineSpecification.rotors[0]

Endif

Endfor

For (ring in Details.rings)

If (1 > ring or 26 < ring)

ErrorReadout.write(“invalid ring position (1-26)”

ring = 1

endif

endfor

for (position in Details.positions)

if (position not integer)

convertToNumber(position)

endif

if (1 > position or 26 < position)

ErrorReadout.write(“invalid position (1-26)”)

Position = 1

Endif

Endfor

If len(details.plugboard) not even

ErrorReadout.write(“lone plug in plugboard”)

Details.plugboard = empty;

endif

Else

For (letterpair in details.plugboard)

If (not letterpair.first in alphabet or not letterpair.second in alphabet)

errorReadout.write(“invalid plug”)

details.plugboard.remove(letterpair)

endif

endfor

endelse

Machine.setRotors(details.rotors)

Machine.setRings(details.rings)

Machine.setPositions(details.positions)

Machine.setPlugs(details.plugboard)

Visualise()

endFunc

This structure affords me two important features. It allows me to treat any complicated encryptions and machine simulation/states as a black box of underlying C++ code, modularizing code to allow expandability and easy to change one part without affecting the other. This structure also allows me to do any validation in python, as C++ validation is more complicated and far harder to debug, and would probably lead to frequent crashing on unexpected inputs.

#### Key Variables and Objects

There will be some key variables and objects to keep track of in Python. The most important of which will be Tkinter’s Tk() object, named in this program as root. This will keep track of the gui and currently open window, and will call all of the programs other functions through its button function calls.

Another important object will be MachineSpecifications. This will store an array of C++ MachineSpecification objects. Each of these will store information about a specific variant of the enigma machine and its wirings, and must be loaded at boot from a JSON file. The user can then select the variant they want to simulate through a dropdown menu of currently loaded specifications.

Beyond this, an important global object will be the Machine, an object containing all the necessary information and underlying objects to simulate an Enigma Machine. This will be used in the simulation tab of the program, and can be updated and drawn from to store the state of the program. Important fields will include a pointer to the currently loaded MachineSpecification, the currently stored Rotors, the reflector and the plugboard.

Finally, the last important global object will be the SolverMachine. Being a derived class of Machine, this also stores the state of the currently loaded machine however will have less getters as it will not be displayed visually. Instead, SolverMachine will contain various methods to cry and decrypt a given Enigma Machine given an unknown ciphertext. Outside of the fields used in Machine, this will contain a “home position” field, storing the best known position of the Machine, allowing it to revert to this state before checking a new setting.

### User interface

#### Simulator Tab

Chart

Description automatically generated

Above is a wireframe of the Simulator tab interface. It can be split into 3 parts: the visualizer (left), the details selector (right), the string input (below)

##### Visualiser

Consisting mostly of standard windows buttons, the data input for this part of the system will be very simple. Simply incrementing or decrementing a value and handling wraparound is quite simple. However, the difficulty here will come from the data output. First, data will be retrieved from the underlying Machine class on how to draw wires. Next a decision will be made based on the machine type loaded whether to draw the 4th rotor with buttons and settings or to make it a simple passthrough (ie A-A, B-B, C-C with no buttons) . It is more convenient to make the rotor a passthrough as this allows the width of the window and number of rotors to be fixed, reducing the amount of work needed to change the windows aspects. Next the buttons and settings options will be handled by tkinter’s native support for these standard windows text in and out as well as standard windows buttons. Next the wirings will be drawn. Drawn from the machine data, each wire will be represented by a line going between known x-values and y-values determined by the offset of their letter. Finally, ontop of this and drawn similarly will be the currently encrypted letter route. If the user presses a button on their keyboard, the underlying class will provide a list of points this letter will go through, allowing the visualizer to draw this route to allow the user to understand how the enigma machine worked.

Func button\_pressed(button)

Machine.update\_setting(button)

Visualize()

endFunc

Func Visualise()

rotorIds.update(Machine.get\_rotorIds())

ringSettings.update(Machine.get\_ringSettings())

reflectorId.update(Machine.get\_reflectorID())

root.delete\_lines()

for (wiring in Machine.get\_wirings())

for(wire in wiring)

drawLine(black,

wiring.leftxOffset,wire[start].yOffset ->

wiring.rightxOffset,wire[end].yOffset

)

Endfor

Endfor

Endfunc

Func keyPressed(key)

If keypress in alphabet

Visualise()

Path = machine.encryptletterverbose(key)

For I = 0 -> 6

drawLine(blue

xoffset[7-I],path[I].yoffset ->

xoffset[6-I],path[I+1].yoffset

)

endfor

for I = 0 -> 6

drawLine(yellow,

xoffset[I], path[8+I].yoffset ->

xoffset[I+1], path[9+I].yoffset

)

Endfor

Endif

Endfunc

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data In | Expected result |
| Rotor Buttons | Test for rotor buttons and wiring | Click rotor buttons and scroll through | Wirings and rotor ID change in accordance |
| Ring buttons | Test for ring buttons and wiring | Click ring buttons and scroll through | Wirings and ring number change in accordance |
| Position buttons | Test for position buttons and wiring | Click position buttons and rotate | Wirings rotate through |
| Key press | Test to see if wires light up | Press a valid key on keyboard | A path lights up (blue forward, yellow back) |
| Key press | Test to see if verbose encrypt works | Press a valid key on keyboard | CORRECT path lights up (blue forward yellow back) |
| Reflector buttons | Test for reflector buttons | Click reflector button | Reflector functions differently/ID changes |
| Key press | Validation | Invalid key pressed | Nothing |
| Key press | Test for key press to be invalidated by text box selection | Type into a text box on window | Nothing |

##### Details Selector

The details selector will be a tkinter box with grey background. It will contain 4 parts, stacked on top of each other. The top part will select the machine type. This will consist of a dropdown menu containing the names of each loaded machine specification, allowing quick changes for the user. Changing this menu – unlike other parts of the detail tab – must immediately change the visualizer on the left, to reduce confusion for the user.

Below will be the machine details input. The Rotors selection will consist of 3-4 dropdown menus, allowing the user to select from a list of available rotors (dependant on the specification loaded). Next will be a number selector, allowing ring settings to be typed in manually or scrolled through with buttons. Unlike ring settings, position settings will be input manually and unable to be scrolled through, consisting of small text boxes. This allows the user to input both numbers and letters, inline with the various formats of enigma messages present across the internet. There will then be a text box for the plugs. This textbox will ignore spaces but require letter only input, allowing the user to input a plugboard either pair by pair with spaces or in one long string of pairs. The position, plugboard and to a lesser extent ring settings must be validated before being sent to the machine.

Below these are the two control buttons (both being simple windows buttons). This will either allow the user to reset the settings to home position (I II III, 1 1 1, A A A) or send them to the machine. Pseudocode for this process has been included in the Structure section of the design as an example.

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data in | Expected Data |
| Valid Data | General test for functionality | Valid Settings | Visualiser updates accordingly |
| Machine Type | Test to see if machine type selector works | Change from Enigma I to M3 | Visualiser updates, more rotors available |
| Machine Types load | Test if specifications have loaded | Click drop down menu | Variants I, M3, M4 are available |
| 4 Rotor Type | Test to see if program correctly handles switch to 4 rotors | Change from Enigma I to M4 | Visualiser updates,  More rotor options |
| Invalid Data | Test to see if error readout is accurate | (individually and combined) invalid settings | Relevant error readout |

##### String Input

The string input part of the simulator tab will be the simplest one. Taking in a string provided by a standard windows text input box, the program will automatically output the encrypted version of the text through a standard windows text output box. The state of the machine will be unaffected by this encryption, and will use the currently loaded state shown by the visualizer (not the details selector).

Func onTextInputUpdate()

String = textbox.getText()

In = clean(string)

Machine.saveState()

Result = Machine.encryptWord(in)

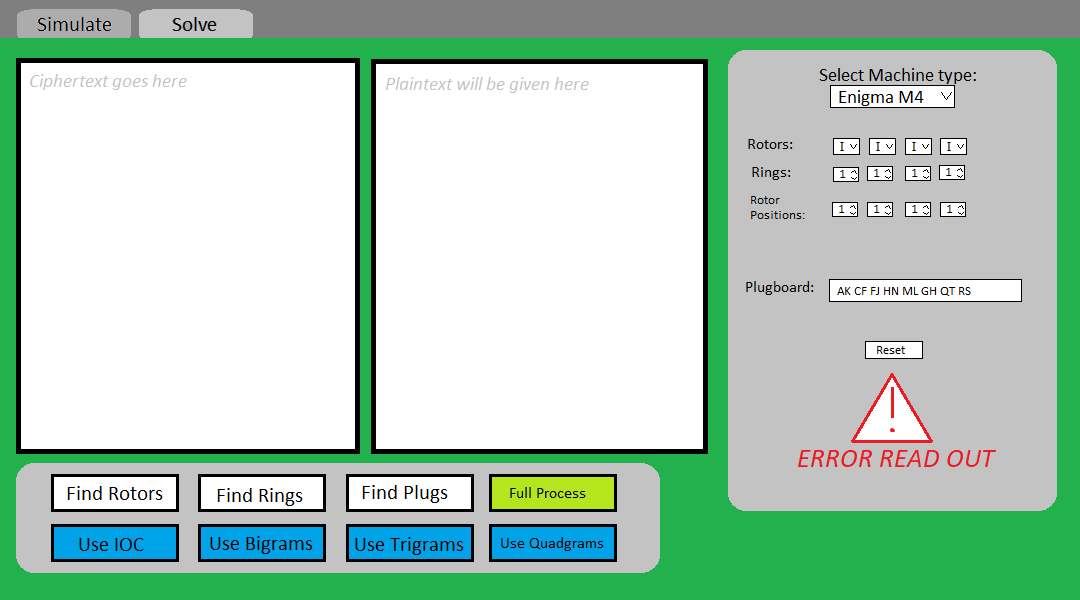
Machine.loadState()

Outputbox.setText(result)

Endfunc

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data In | Expected Data |
| Valid data | Test to see if program handles simple input | Full capitals input | Any output in result text box |
| Valid data | Test if program correctly handles simple input | Full capitals input | Correct output in result box |
| Valid data | Test for lower case handling | Mixed capitals/lower case input | Correct full capitals output in result |
| Borderline data | Test for non-alphabet handling | Mixed capitals/lower case with numbers and punctuation | Correct full capitals output, with non-alphabet characters ignored |
| Invalid data | Test for no valid input | Numbers and punctuation alone | Error read out should give warning, no output |

#### Solver Tab



Above is a wireframe of how the solver tab will appear. Of note is the text boxes, button panel and selection section. It will largely copy the selection section from the simulation tab so I will not re-explain entirely here.

##### Text Boxes

The text boxes in this tab are quite simple and will not update live. The left ciphertext box will be an input box, allowing the user to input ciphertext which will be sent to the solver machine when any of the buttons are pressed. There will need to be similar input validation as the string input, and so shall use the same testing.

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data In | Expected Data |
| Valid data | Test to see if program handles simple input | Full capitals input | Any output in result text box |
| Valid data | Test if program correctly handles simple input | Full capitals input | Correct output in result box |
| Valid data | Test for lower case handling | Mixed capitals/lower case input | Correct full capitals output in result |
| Borderline data | Test for non-alphabet handling | Mixed capitals/lower case with numbers and punctuation | Correct full capitals output, with non-alphabet characters ignored |
| Invalid data | Test for no valid input | Numbers and punctuation alone | Error read out should give warning, no output |

##### Details Selector

Borrowing largely from the details selector of the simulator tab, the machine type selector will be identical, and the tab will follow the same layout. It will differ however in the lack of send to machine button (replaced by the control panel), and the program will automatically update the settings as the encryption continues. This allows me to re-use code and formatting to make an intuitive, coordinated design for the whole program.

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data in | Expected Data |
| Valid Data | General test for functionality | Valid Settings | Visualiser updates accordingly |
| Machine Type | Test to see if machine type selector works | Change from Enigma I to M3 | Visualiser updates, more rotors available |
| Machine Types load | Test if specifications have loaded | Click drop down menu | Variants I, M3, M4 are available |
| 4 Rotor Type | Test to see if program correctly handles switch to 4 rotors | Change from Enigma I to M4 | Visualiser updates,  More rotor options |
| Invalid Data | Test to see if error readout is accurate | (individually and combined) invalid settings | Relevant error readout |

##### Control Panel

This panel will consist of 8 standard tkinter buttons arranged in a grid format. Each button will have its own process linked to it in the underlying solver machine, and must load the current settings from the details selector and ciphertext from the ciphertext box, validate and then send to the solver machine, performing an attack specific to the button pressed (and described in the next section) before taking the resultant settings and updating the details tab, followed by encrypting the ciphertext with these given settings and updating the result box. The 4 bottom buttons will not follow this procedure however, only being used to change the analyser that the solver machine is using for its attacks.

Func onButtonPress(button)

If button.requiresDetails()

Ciphertext = textin.get()

In = clean(ciphertext)

Details = getDetails()

Valid = details.validate()

If valid

solverMachine.update(details)

solverMachine.doAttack(button,in)

resultDetails = solverMachine.getResultDetails()

setDetails(resultDetails)

result = solverMachine.encryptWord(In)

resultbox.setText(result)

endif

else

solverMachine.updateAnalyser(button)

endif

endfunc

Testing for this section will mostly be covered under testing for the underlying functionality, but a range of valid and invalid inputs should be tested for crashing or invalid outputs.

### Underlying Functionality

Tests in this section will be more specific, but contain less validation as these tests will be more abstract, often being used during development to check for bugs in the functionality as opposed to actual inputs expected of the final program.

#### Simulating

The simulation of the enigma machine will be all wrapped in C++, allowing the code to be re-used in the solver with no performance loss, while also abstracting it away from the GUI sided python code.

##### Mutate Rotors

The Machine class will contain a set of rotor objects, each of which contains information about the wirings of the rotor, the position of the rotor, the place of the rotor and the ring of the rotor. This allows each rotor to be its own self contained object and mutation to be broken down into evaluating the new position of each rotor.

The process of mutating the rotors itself is simple. The right rotor rotates every key press, and as such its position will increment every mutation. The middle rotor rotates in one of two conditions, either the right rotor is in the turnover position, or the middle rotor itself is in the turnover position. This leads to a behaviour where after the right rotor rotates the middle into the turnover position, the middle rotor turns a second time. This is called “double stepping” and links into the left rotor, which turns whenever the middle rotor is in the turnover position. If the middle rotor did not double step, it would remain in the turnover position until the right rotor passes again, making the left rotor rotate through every step like the right rotor and decreasing the period from 26 \* 25 \* 26 to 26 \* 26. The fourth rotor never rotates.

Class Rotor

Public func mutate()

If rotor.place = right

Position = normalise(position + 1)

Elif rotor.place = middle

If rightRotor.inTurnover()

Position = normalise(position + 1)

Elif rotor.inTurnover()

Position = normalize(position + 1)

Endif

Elif rotor.place = left

If middlerotor.inTurnover()

Position = normalise(position + 1)

Endif

Endif

Endfunc

endclass

Func MutateRotors()

rightRotor.mutate()

middleRotor.mutate()

leftRotor.mutate()

endFunc

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data in | Expected Out |
| Simple Mutation | Simplest state | Mutate I, II, III, 1, 1, 1 | 1, 1, 2 |
| Random Mutation | Random simple state | Mutate VII, I, V, 3, 6, 16 | 3, 6, 17 |
| Right Rotor Step | Right rotor on turnpoint | Mutate III, V, II, 22, 25 5 | 22, 26, 6 |
| Middle rotor step State | Middle rotor on turnpoint | Mutate III, V, II 22, 26, 5 | 23, 1, 6 |
| All rotor step state | Middle and right rotor on turnpoint | Mutate III, V, II, 22, 26, 6 | 23, 1, 7 |

##### Encrypt Letter

The encryption of a letter in the enigma machine is symmetric, meaning every letter goes on the same journey forwards as backwards in the machine. This is explained by the encryption of a letter, following the path of Keyboard -> plugboard -> right rotor -> middle rotor -> left rotor -> reflector -> left rotor -> middle rotor -> right rotor -> plugboard -> light. Of note is that the text takes the reverse route after the reflector, so if A->B and B->C on the forward direction, in reverse B->A instead of B->C.

In the code, substituting a letter through these steps will be made quite simple by the usage of simple substitution arrays, requiring only an indexing to get the substituted letter forward, and backward substitution can be done by creating and storing a reverse substitution array.

Func EncryptLetter(int in)

mutateRotors()

plugForward = plugboard[in]

rightForward = rightRotor[plugforward]

middleForward = middleRotor[rightForward]

preReflector = leftRotor[middleForward]

if (rotor.is4Length)

preReflector = 4thRotor[preReflector]

endif

postReflector = reflector[preReflector]

if (rotor.is4Length)

postReflector = 4thRotorReverse[preReflector]

endif

leftBackward = leftRotorReverse[postReflector]

...

plugBackward = plugboard[rightBackward]

return plugBackward

endfunc

Various substitutions will be tested in this table, but I have left the whole algorithm to be tested in the modes of encryption section. Important to note is that all of the forward rotor transformations will be post mutation, so position will usually be 1 more than the given. This is to line up with the online enigma machine being used to test this. However, the backward transforms will not be post mutate, as on the online machine I will use the leftmost rotor to test which does not rotate

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data In | Expected |
| Direct Substitution | Using wirings from Reflector B | A | Y |
| Direct Substitution | Using wirings from Reflector B | N | K |
| Direct Substitution | Using wirings from Reflector B | Z | T |
| Rotor Forward | Rotor I wirings | Pos 1, ring 1, A | J |
| Rotor Forward | Rotor III wirings | Pos 7, ring 4, E | N |
| Rotor Forward | Rotor V | Pos 25, ring 2, P | J |
| Rotor Forward | Rotor VIII | Pos 26, ring 26, Z | E |
| Rotor Backward | Rotor I | Pos 2, ring 13, P | L |
| Rotor Backward | Rotor III | Pos 23, ring 20, L | W |

##### Modes of Encryption

There will be 2 distinct ways of encrypting in this system, the simpler being to encrypt an entire word (used in the string input box and solver sections) or to encrypt 1 letter and return the state of the machine at every point, allowing visualization. To illustrate the algorithm in full for both modes, here are their flow charts.

Diagram

Description automatically generated

Diagram

Description automatically generated

With this out of the way, some concrete tests can be established to evaluate the entire system.

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Details | Data In | Expected |
| No plug Encryption Verbose | Simple test | VII, V, II, Position 24, 4, 17, Ring 8,6,3, letter T | T, T, I,V, G,G, L,L, W, K, D, D |
| No plug encryption verbose | Simple test | VII, IV, III, position 19, 23, 18, ring 11, 26, 8, letter V | V,V,R,A,E,E,Q,Q,B,L,G,G |
| No plug encryption verbose | Simple test | I, III, II, position 1, 22, 25, ring 13, 20, 8, letter H | H,H,M,B,I,I,P,P,L,W,G,G |
| No plug phrase encrypt | Check as many letters as possible | III, V, VII, position 9,11,12, ring 2,5,20, "LAZYDOGJUMPS" | DLQHHJOWPDHM |
| No plug phrase encrypt | Check as many letters as possible | VI, I, VII, position 11,24,2, ring 21,18,25, "QUICKBROWNFOX" | ZFTKDXSHQJZSM |
| Plugged phrase encrypt | Simple input | I, II, III, position 1,1,1 ring 1,1,1, plugboard "AB CD EF GH IJ KL MN OP QR ST", "HELLOWORLD" | XLNGIBJPTF |
| Plugged phrase encrypt | Check as many letters as possible | VII,II,V, position 2,19,18 ring 25, 7, 12 plugboard "EX ST FP AN", "LAZYDOGJUMPS" | KQWSIIDCDTYZ |
| 4 Machine plugged phrase encryption | Encrypted phrase will be full “quick brown fox text” | Beta, IV, V, VIII, position 3,17,23,11 rings 8,9,3,4 plugboard "JK FR BE IP ZS AW", "QUICKBROWNFOX" | JFVKFEBWKTRTS |

#### Solving

Solving the enigma cipher, as described in the paper at <https://web.archive.org/web/20060720040135/http://members.fortunecity.com/jpeschel/gillog1.htm> is quite a simple process, making use of the fact that the enigma machine produces large stretches of correct text, even when ring and plugboard settings are wrong. This allows a bruteforce attack to be performed to find the correct series of rotors and rotor positions, taking a short amount of time to search the space of possible keys on modern hardware and finding the one that resembles clean text best. This can be followed by searching the entire space of possible ring settings (only relevant to middle and right rotors as the ring settings essentially only take effect when they step the next rotor). Finally, plugboard settings are found by looking at every possible combination of two plugs and adding the highest scoring one, repeating until there are no more plug combinations which lead to cleaner text.

Func findBestRotors()

Maxscore = 0

For rotorCombination in rotors

For positionCombination in [1-26]

Machine.setRotors(rotorCombination)

Machine.setPositions(positionCombination)

Result = Machine.encryptWord(ciphertext)

Score = analyse(result)

If score > maxscore

Maxscore = score

Maxsettings = rotorCombination,positionCombination

Endif

Endfor

Endfor

Machine.setSettings(maxSettings)

Endfunc

##### Analysis

To evaluate whether a potential cleantext is close to real language, a few algorithms are used. An emphasis is placed on the algorithm being fast as this algorithm will be called millions of times in the rotorFinding step. The two algorithms described by the paper are Index of Coincidence and NGrams.

The simpler of these is Index of Coincidence and is described by this formula Text

Description automatically generated with medium confidence, requiring the suspected cleantext be searched through once to find the frequency of each letter in the text, then these frequencies be used in this calculation. In the final code, I will not use the N(N-1) part of the formula, as every word will be the same length (owing to the fact they all have the length of the same ciphertext), adding a costly divide to the process.

The other, NGrams, requires an N be described (between 2-4) and is very similar to the analysis technique used on simple substitution ciphers. First, a large portion of text of the language being tested gets broken down into sequences of length N. These sequences are then counted and their frequencies taken a logarithm. This can be done ahead of runtime and stored in a file. Then at runtime, these frequencies are loaded and put in a large table. Then, when analysing the potential cleantext is also broken down into N sized sequencies, and these sequences are looked up in the table and summed. Due to the properties of logs, this is equivalent to taking the probability that this cleantext would be produced in a piece of text. Due to the fact this requires lookups and breaking the text down, this is a more costly analysis that I will use for the faster ring and plugboard sections but not the rotor part.

## (iii) DESCRIBE THE APPROACH TO TESTING

1. *Identify the test data to be used during the iterative development and post development phases and justify the choice of this test data.*

The system will be tested in many ways. The first that will be used is to test the functionality of the underlying functional code using a compiled debug executable. This will allow me to adapt and test the code iteratively while working on the functionality, as opposed to having to connect the C++ code to python and a GUI. The functional code will not have any validation, so no invalid or borderline testing will be done here, only testing for functionality, using bigger and bigger tests to cover more and more failure points.

Tests for the python frontend have been described in the preceding section.

3 sources are required for testing and specifications throughout the project.

<https://www.cryptomuseum.com/crypto/enigma/wiring.htm> - This provides wirings in plaintext format about each type of enigma machine, and is an archive of known enigma wirings and specifications.

<https://piotte13.github.io/enigma-cipher/> - This will allow me to look at the path of an enigma message as it passes through the machine, allowing me to see troubleshoot bugs in my program by seeing exactly where it fails to line up with expected enigma translation. This will not be as useful when troubleshooting the M4.

<https://people.physik.hu-berlin.de/~palloks/js/enigma/enigma-m4_v16_en.html> - This program emulates an Enigma M4 with a large input text box. While this is not as useful for testing single letters as its route visualisation is less than ideal, this allows me to generate large amounts of ciphertext to test the solver. Also of note is that, while this emulates an Enigma M4, this can be used to emulate the M3 and I by leaving the 4th rotor in position 1

Finally, the system will be tested by each of my stakeholders and I shall give them a questionnaire to evaluate how well the program has explained their understanding both of how the enigma machine worked and how the machine is solved by the bruteforce algorithm. To test the decryption feature, I will allow them to use one of the previously specified other applications which implement the enigma cipher.

# (3) DEVELOPING THE SOLUTION (25 mARKS)

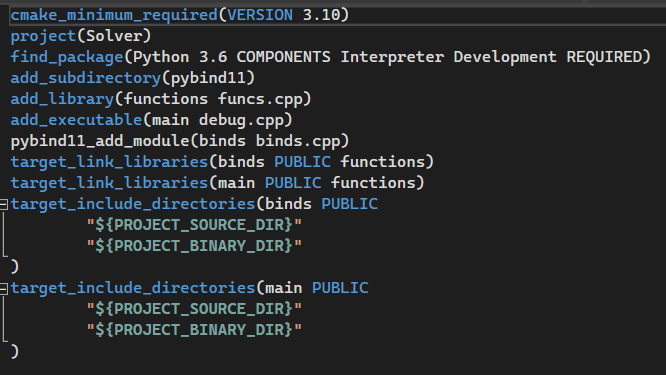
## (i) INTERATIVE DEVELOPMENT PROCESS

*(a) Provide annotated evidence of each stage of the iterative development process justifying any decision made.*

*(b) Provide annotated evidence of prototype solutions justifying any decision made*.

I began development by establishing the directory tree, pybind11 and CMake, essentially settings up my environment and build tools. Originally the file structure looked like src/Solver/Cpp/(C++ development) however as development progressed I realized that as C++ grew in complexity, it would require a better environment, leading to the final structure of src/Cpp/(C++ development)

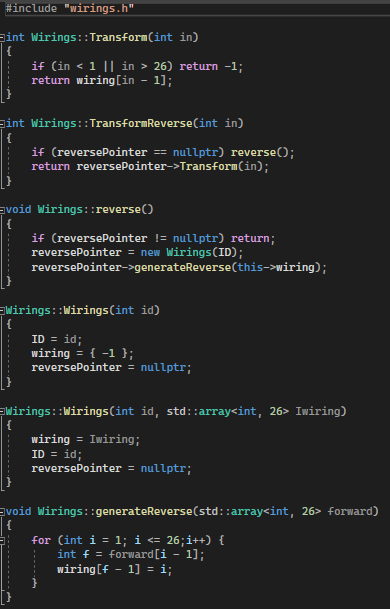
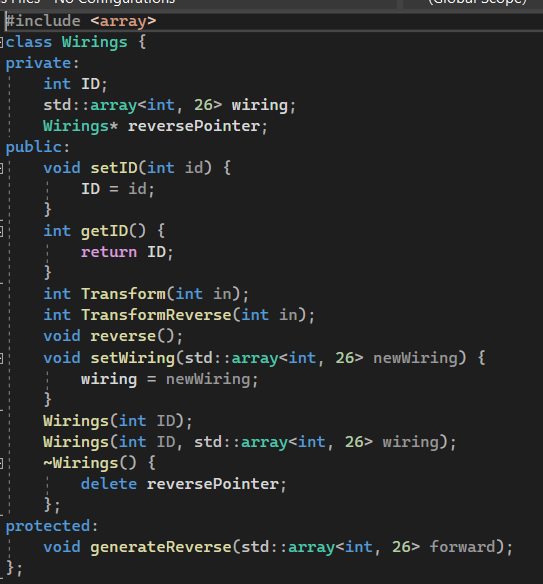
The CMake file during the beginning of development was simple, and can be seen below.



This file, in order, initializes cmake and the cmake project, finds pybind11 which is submoduled into the project folder, adds the library funcs.cpp, creates an executable from debug.cpp and creates a python binding which can be imported into python. The rest simply links functions into debug and bindings so they can be accessed there. I would leave binds as an empty file until the end of C++ development, and debug would change rapidly throughout development. Funcs is not used again, I simply used it to test the CMake project structure. Throughout development, I would link files and do various small changes to CMake not documented here such as create sub folders containing their own CMake libraries.

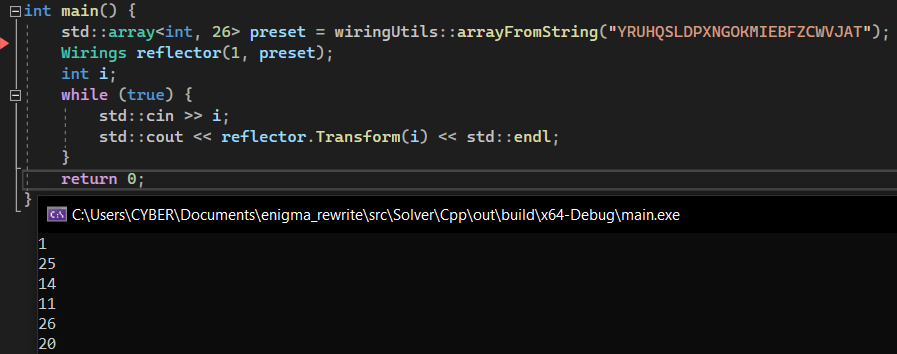
The first code I created was the wiring class. This would be a vital class for the project, as it serves as a base class for almost every important other object. This is because the plugboard, reflector and rotors are all essentially just substitutions, which can all be wrapped in this “Wirings” class to provide safety and quick changes.

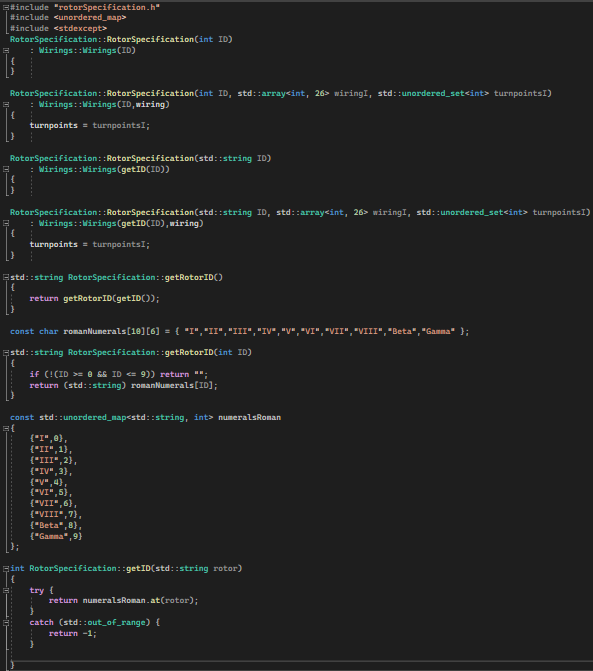
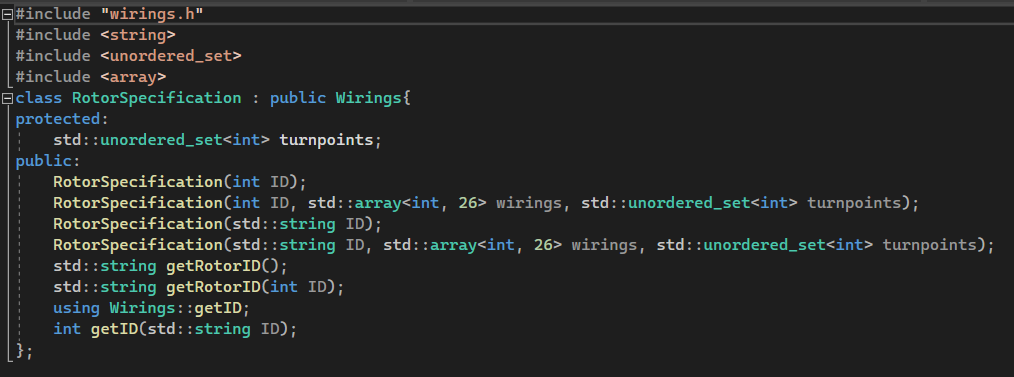
Here can be found the header and source files for wirings:



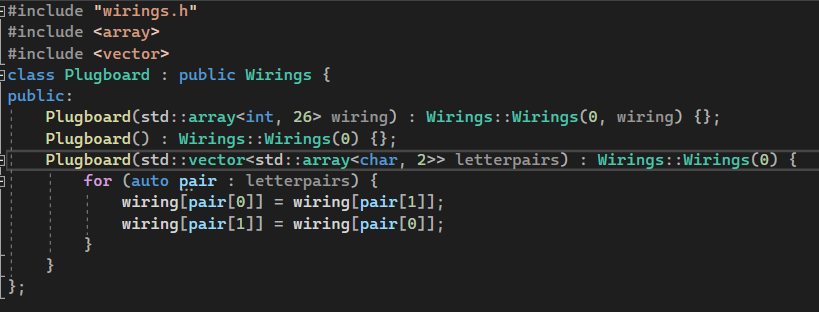
Of note is that I decided to make the class contain a pointer to another object of the same class. This “reversePointer” would be generated in the generateReverse() function and be automatically called by reverse() and contain the reverse of the wirings given. This essentially allowed me to cache the reverse of the wirings, greatly speeding up the bruteforce attack later. Also of note is providing the class with an integer ID. This would not be used often, but provided some debugging tools and uses when loading the real names of the classes derived from wirings. Using this class I could test the following table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Details | Data In | Expected | Result |
| Direct Substitution | Using wirings from Reflector B | A | Y | 25 (Y) |
| Direct Substitution | Using wirings from Reflector B | N | K | 11 (K) |
| Direct Substitution | Using wirings from Reflector B | Z | T | 20 (T) |

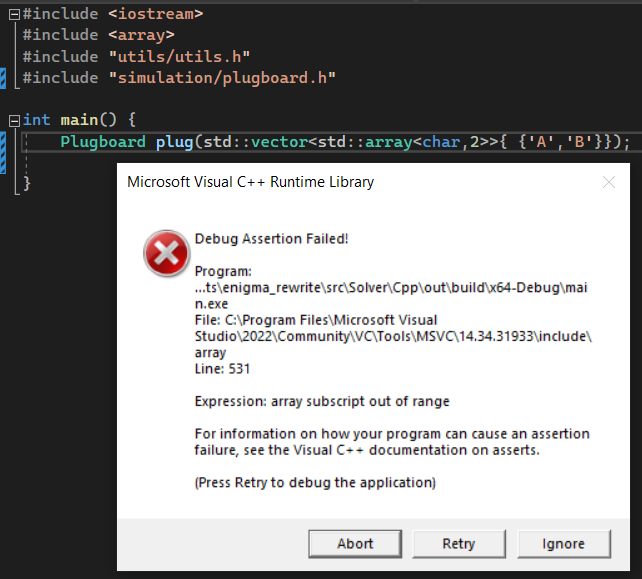


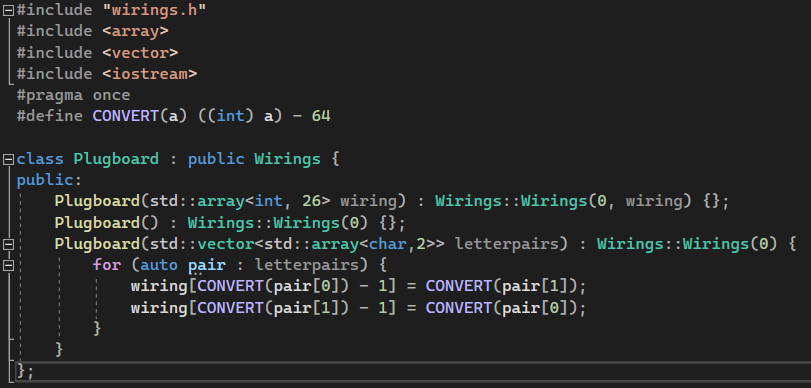
I then implemented the reflector class, inherited almost entirely from wirings. For conciseness I have not included this. 

Work then began on the rotorSpecification class. Consisting mostly of constructors and methods to get the name of a rotor, the rotorSpecification class contains the wirings of a rotor without any information about its state. This allowed me to build rotors quickly by copying a rotorSpecification and abstracted away some information that would have to be stored if a machineSpecification stored the rotor wirings with a more procedural approach. Of note is the usage of an unordered\_set to store the turnpoints of a rotor. This allowed me to insert and search through arbitrary amounts of turnpoints on any one rotor. I then worked on the plugboard class

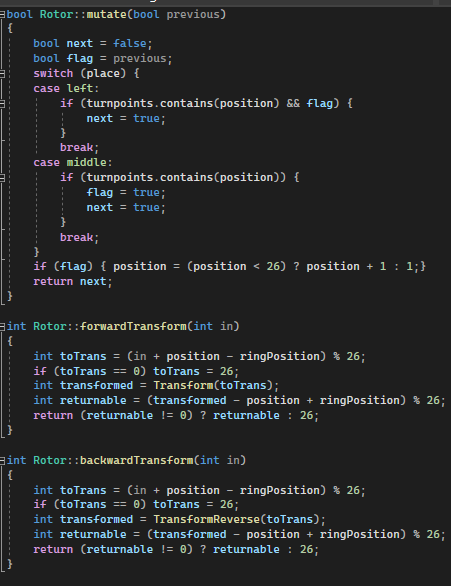
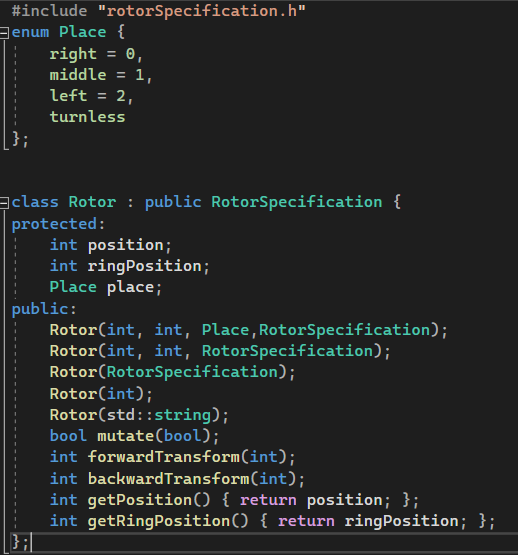


I thought this would be quite a simple class to design similar to the reflector, however this code ended up being very buggy. If any character would be input to the plugboard in the letterpairs, it would immediately crash as I forgot to convert the char to an integer from 1-26.



To fix this, I defined a macro called CONVERT which would convert a char to a 1-26 int 

I then created the beginnings of the rotor class, inheriting from the rotor specification class to borrow its wirings, id and turnpoint features.



With the implementation of rotor came the implementation of the mutate function and the transforms. I decided to make the mutate function take in a Boolean denoting whether it should turn and return a Boolean determining whether the next rotor in the sequence should turn. This allows each rotor to independently evaluate its own position (abstracting it away from a bigger method). Transforms are done by the shown method. Adjust for rings and position, normalize to the expected range then send to the underlying Wirings transform, before then correcting again and normalizing. I did however find the usage of modulus inefficient in this context, replacing it with a Wirings::normalize function which would use if statements to correct the values for increased performance, as I knew the expected values should never be more than 2 corrections away in normal usage.

Text

Description automatically generated

With the rotor class established, I could now test this table fully

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Details | Data In | Expected | Result |
| Direct Substitution | Using wirings from Reflector B | A | Y | 25(Y) |
| Direct Substitution | Using wirings from Reflector B | N | K | 11(K) |
| Direct Substitution | Using wirings from Reflector B | Z | T | 20(T) |
| Rotor Forward | Rotor I wirings | Pos 1, ring 1, A | J | J |
| Rotor Forward | Rotor III wirings | Pos 7, ring 4, E | N | N |
| Rotor Forward | Rotor V | Pos 25, ring 2, P | J | J |
| Rotor Forward | Rotor VIII | Pos 26, ring 26, Z | E | E |
| Rotor Backward | Rotor I | Pos 2, ring 13, P | L | 14(N) |
| Rotor Backward | Rotor III | Pos 23, ring 20, L | W | 4(D) |

**Text

Description automatically generated**

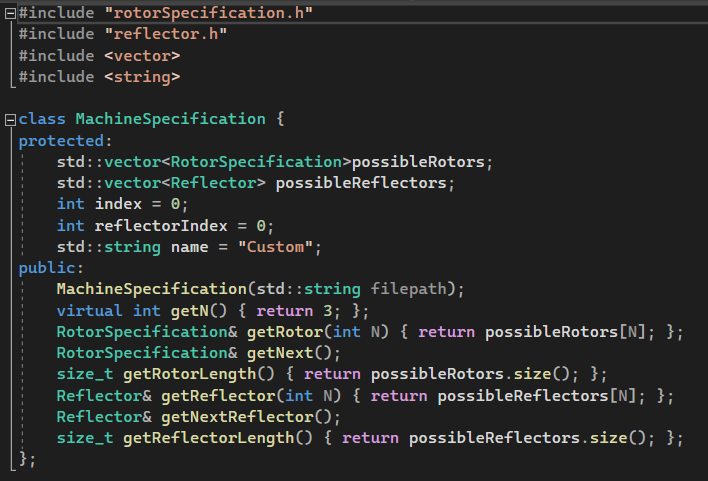
What I realized after spending some time looking at the original website to see how this proceeded, I realized that my testing apparatus was at fault, as opposed to the rotor code itself. The error comes from mutating before the backward transforms, when in reality they should be fixed. Adjusting this, the code is found to work immediately.

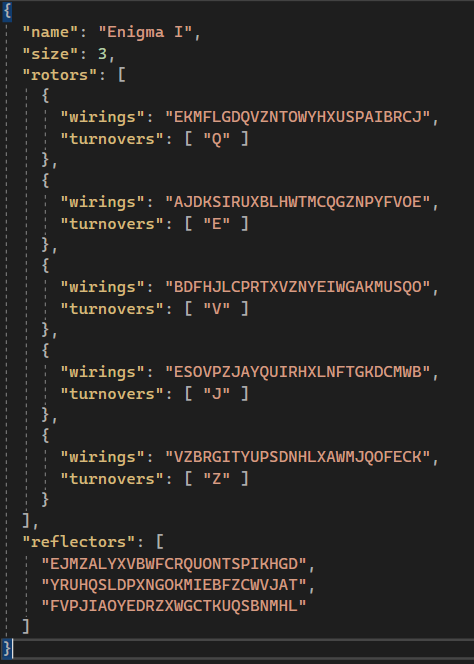
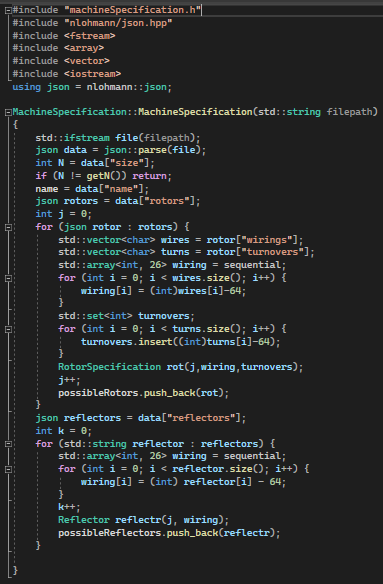
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rotor Backward | Rotor I | Pos 2, ring 13, P | L | 12(L) |
| Rotor Backward | Rotor III | Pos 23, ring 20, L | W | 23(W) |

Text

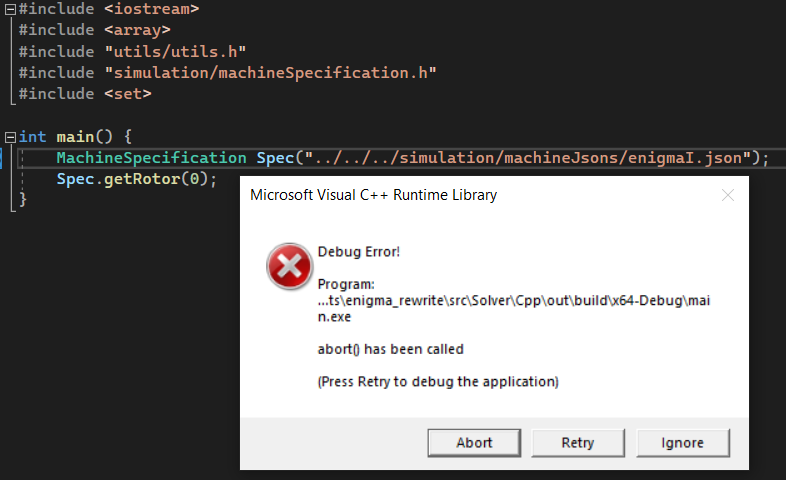
Description automatically generated

Having realized how complex and unnecessary this testing was with the objects as they stand, I resolved to create the machine simulation class, which would hopefully make the creation of rotors substantially easier.

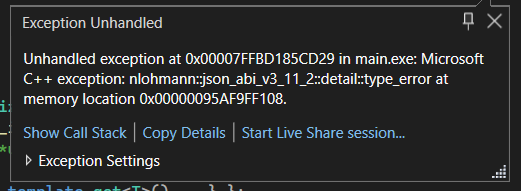
Text

Description automatically generated

The machine specification makes use of nlohmann’s JSON C++ library to load JSON files containing information about the rotors, their wirings and turnpoints. Each rotor is a json container with wirings and an array of turnovers. These are stored in the rotors array, and the reflectors are stored as their wirings in an array. The format makes more sense with an example JSON file, and I have provided the Enigma I specification JSON.



This code did not work however, calling an unknown abort. By using the Visual Studio debugging tools I tracked this error down to the constructor loading the turnovers and wires of the rotors.

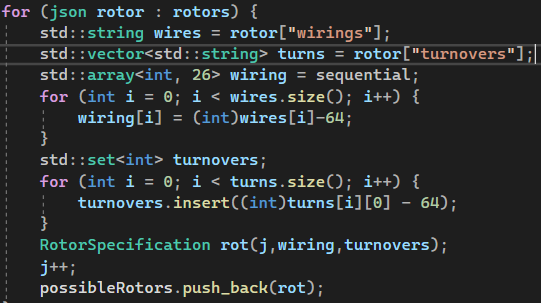
Text

Description automatically generated

Text

Description automatically generated

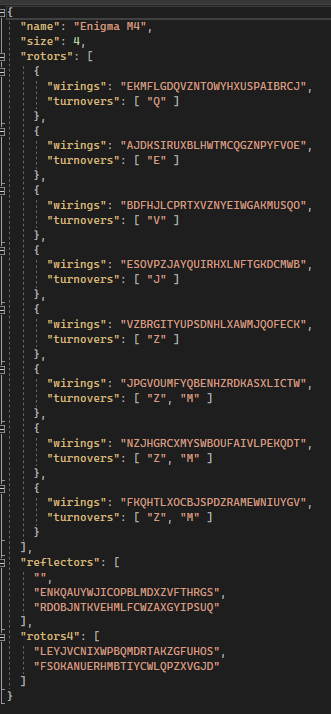
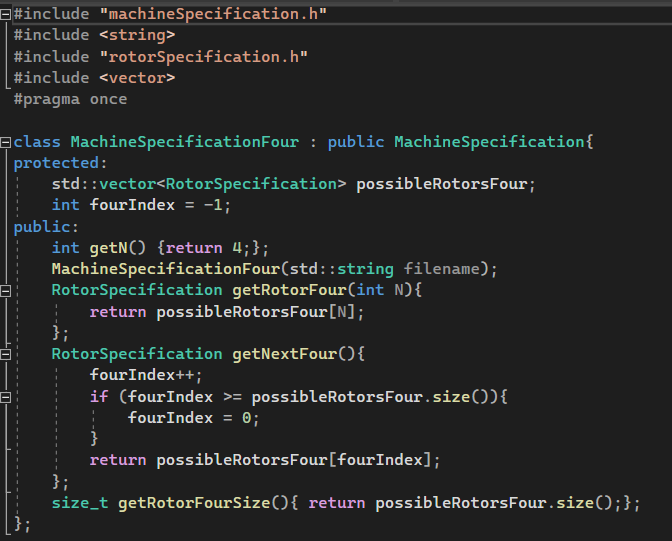
This would fail as the JSON library didn’t know how to convert a vector of characters into a string, and would create every letter in the turnovers as its own string, as opposed to a char as I expected. This would be quickly resolved by changing the first vector to a string and the second to a vector of strings.



This resulted in the following, accurate, output.

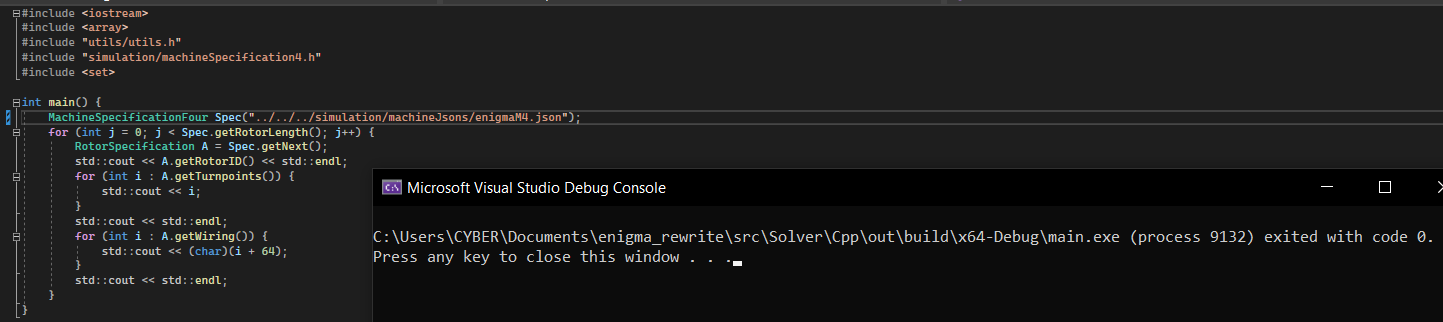
Text

Description automatically generatedThis would be rounded out by implementing the specification for machines of length 4.

Text

Description automatically generated

Inheriting from MachineSpecification, this class would call the MachineSpecification constructor to grab all the normal rotors and reflectors, but would then re-open the file and read the data about the 4th rotors, stored similarly to the reflectors as opposed to the fully fleshed out normal rotors.

However, upon testing this the program would not give the expected output, instead opting to fully run without crashing, implying that the constructor gives no errors.

Using the visual studio debugger again, I found that the MachineSpecification constructor is returning early because it reads the rotor size in the json file (a 4 here) and believes that this is incompatible with the MachineSpecifiication type it’s trying to construct (a 3). This is not remediated by using a virtual function to get the size of the MachineSpecification, as constructors do not call virtual functions like other functions. To make this work, I would have to create an initializer outside of the constructors that can call the correct getN(), however this seems like quite complicated and would take a lot of refactoring so I opted to just remove the check for the length in the machineSpecification

Text

Description automatically generated

With the base of machineSpecification complete, I could implement the machine class. Being the most important class in the project, this would also be the largest C++ class and contain a lot of important methods.

Text

Description automatically generatedImportant methods in this class include all the encrypts, the mutate rotors and the constructor. When structuring the encrypt methods, I decided to make encryptLetter(int) and encryptLetterVerbose(int) both virtual. This allows me to call these functions from the wrappers (such as EncryptWord and all encrypts with characters) and know that the encryption they provide is still accurate in the derived MachineFour where these can be overridden by encrypt methods that handle 4 rotors.

Text

Description automatically generated

Here is the definition of the constructor. This copies a pointer of the specification into the Machine fields, allowing the machine to change its rotors at any time. It then goes on to initialize the rotor positions and indexes with the constructor list and the specification.

Text

Description automatically generated

Allowed by the highly contained rotor mutate method, this part of the code became relatively simple, only having to chain the result of one mutate into the next. Testing this with the rotor mutate table revealed some problems however.

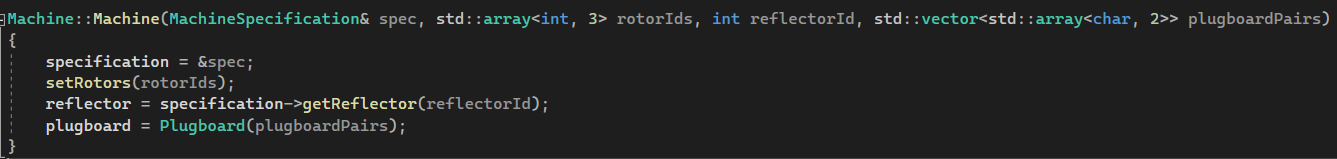
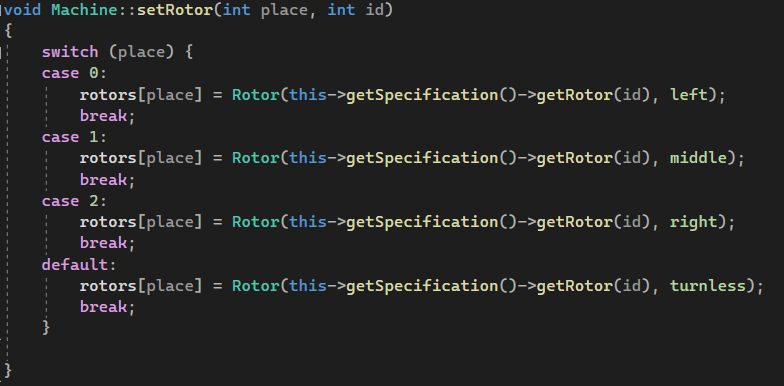
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Details | Data in | Expected Out | Actual result |
| Simple Mutation | Simplest state | Mutate I, II, III, 1, 1, 1 | 1, 1, 2 | 1, 1, 2 |
| Random Mutation | Random simple state | Mutate VII, I, V, 3, 6, 16 | 3, 6, 17 | 3,6,17 |
| Right Rotor Step | Right rotor on turnpoint | Mutate III, V, II, 22, 25 5 | 22, 26, 6 | 23,26,6 |
| Middle rotor step State | Middle rotor on turnpoint | Mutate III, V, II 22, 26, 5 | 23, 1, 6 | 23,1,6 |
| All rotor step state | Middle and right rotor on turnpoint | Mutate III, V, II, 22, 26, 6 | 23, 1, 7 | 23,1,7 |

Text

Description automatically generated

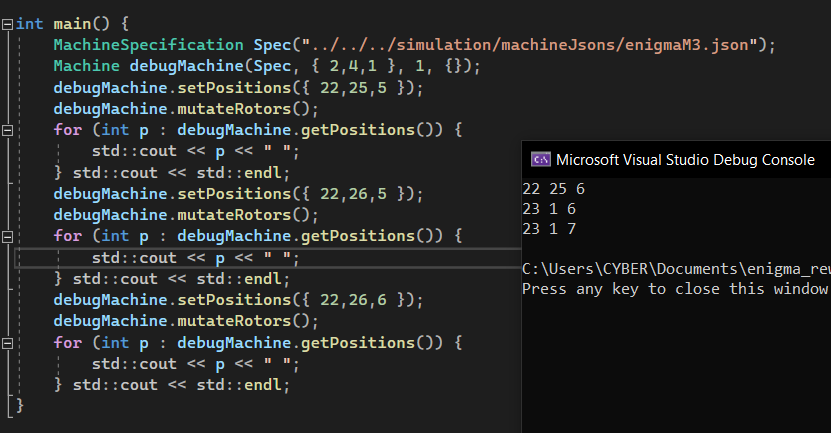
*This was done with every required data state, easier to show one than each setup*

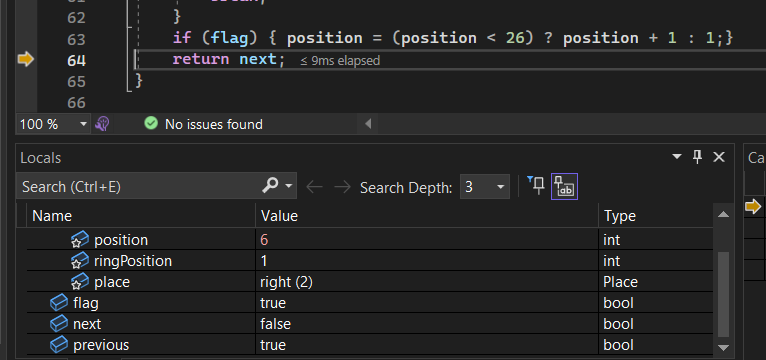
This failure suggested that the leftmost rotor was getting sent the message to rotate, despite the middle rotor not being on the turnpoint. Stepping through the code at this point I noticed a strange discrepancy that each rotor had some wildly different Place fields. This made me realize that the machine class never initialized the Places of each rotor, only producing the default rotor. Why this lead to these having very strange (not in the enum) place values is uncertain, but I addressed this by creating a new method to set rotors as follows.



Retesting, this lead to yet another (different) wrong outcome.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Right Rotor Step | Right rotor on turnpoint | Mutate III, V, II, 22, 25 5 | 22, 26, 6 | 22,25,6 |
| Middle rotor step State | Middle rotor on turnpoint | Mutate III, V, II 22, 26, 5 | 23, 1, 6 | 23,1,6 |
| All rotor step state | Middle and right rotor on turnpoint | Mutate III, V, II, 22, 26, 6 | 23, 1, 7 | 23,1,7 |



Yet again stepping through the code I found that the right most rotor was not returning true when mutating. 

Quickly glancing over my code , I realised my error. I had written left instead of right in the mutate switch method. This would be quickly addressed to reveal the correct behaviour.

Graphical user interface

Description automatically generated with medium confidence

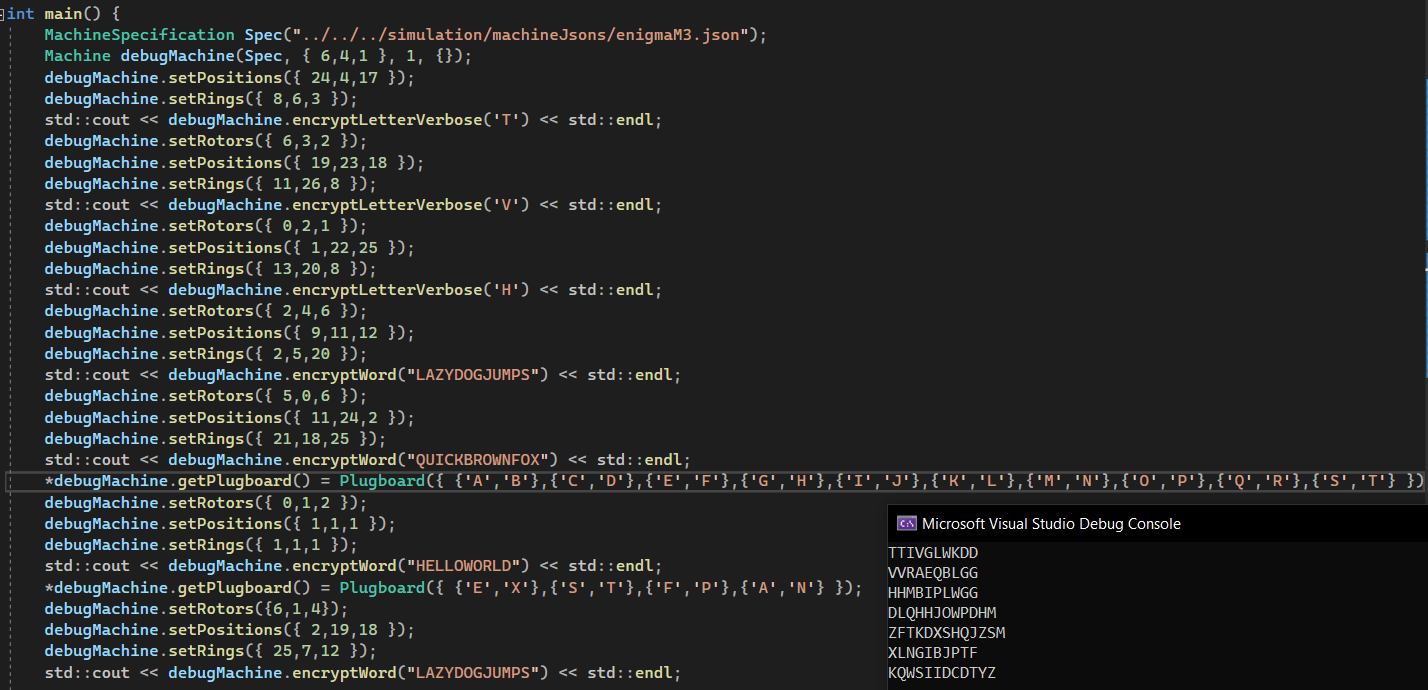
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Right Rotor Step | Right rotor on turnpoint | Mutate III, V, II, 22, 25 5 | 22, 26, 6 | 22,26,6 |
| Middle rotor step State | Middle rotor on turnpoint | Mutate III, V, II 22, 26, 5 | 23, 1, 6 | 23,1,6 |
| All rotor step state | Middle and right rotor on turnpoint | Mutate III, V, II, 22, 26, 6 | 23, 1, 7 | 23,1,7 |

The final part of the Machine class to look at is the encrypt.



Mirroring very closely the flowcharts describing this algorithm, the encrypt algorithms are quite simple. At the start of encrypting a letter, the rotors are mutated. The letter then goes through the series of encrypts, and is returned. There are overloaded functions that work in strings and chars instead of vectors and ints, but these simply call the integer representation after some transforms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Details | Data In | Expected | Actual result |
| No plug Encryption Verbose | Simple test | VII, V, II, Position 24, 4, 17, Ring 8,6,3, letter T | T, T, I,V, G,G, L,L, W, K, D, D | T,T,I,V,G,L,W,K,D,D |
| No plug encryption verbose | Simple test | VII, IV, III, position 19, 23, 18, ring 11, 26, 8, letter V | V,V,R,A,E,E,Q,Q,B,L,G,G | V,V,R,A,E,Q,B,L,G,G |
| No plug encryption verbose | Simple test | I, III, II, position 1, 22, 25, ring 13, 20, 8, letter H | H,H,M,B,I,I,P,P,L,W,G,G | H,H,M,B,I,P,L,W,G,G |
| No plug phrase encrypt | Check as many letters as possible | III, V, VII, position 9,11,12, ring 2,5,20, "LAZYDOGJUMPS" | DLQHHJOWPDHM | DLQHHJOWPDHM |
| No plug phrase encrypt | Check as many letters as possible | VI, I, VII, position 11,24,2, ring 21,18,25, "QUICKBROWNFOX" | ZFTKDXSHQJZSM | ZFTKDXSHQJZSM |
| Plugged phrase encrypt | Simple input | I, II, III, position 1,1,1 ring 1,1,1, plugboard "AB CD EF GH IJ KL MN OP QR ST", "HELLOWORLD" | XLNGIBJPTF | XLNGIBJPTF |
| Plugged phrase encrypt | Check as many letters as possible | VII,II,V, position 2,19,18 ring 25, 7, 12 plugboard "EX ST FP AN", "LAZYDOGJUMPS" | KQWSIIDCDTYZ | KQWSIIDCDTYZ |



This testing result showed that while the algorithm was working fine, I forgot account for 4 rotors when writing the verbose encryption. This was easily remedied by pushing back the letters before and after plugboard twice.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Details | Data In | Expected | Actual result |
| No plug Encryption Verbose | Simple test | VII, V, II, Position 24, 4, 17, Ring 8,6,3, letter T | T, T, I,V, G,G, L,L, W, K, D, D | T,T,I,V,G,G,L,L,W,K,D,D |
| No plug encryption verbose | Simple test | VII, IV, III, position 19, 23, 18, ring 11, 26, 8, letter V | V,V,R,A,E,E,Q,Q,B,L,G,G | V,V,R,A,E,E,Q,Q,B,L,G,G |
| No plug encryption verbose | Simple test | I, III, II, position 1, 22, 25, ring 13, 20, 8, letter H | H,H,M,B,I,I,P,P,L,W,G,G | H,H,M,B,I,I,P,P,L,W,G,G |

Having completed the machine class, the machineFour was the clear next step forward. However, the pitfalls of C++ and object oriented programming began to show their faces.

Text

Description automatically generated

This refused to cast the MachineSpecification cast of Spec to the MachineSpecification required for the Machine constructor. Later in development I’d learn more C++ and realise that there would be a superior method produced by static\_cast<>, however at this point I decided to address the error that the compiler would throw, by removing the call to the Machine constructor and running all the construction in machine4. This would in turn create it’s own error (no default constructor for Machine) but I could remedy this by creating a Machine constructor with no arguments (protected so it can’t be called accidentally).

Also can be seen in this code is that the Specification field uses a unique\_ptr. This is a C++ technique which would handle garbage collection for classes only referenced by 1 pointer, and I thought I would be smart for using it. Later I realised this made no sense, as the MachineSpecification it points to would not be owned by the class, but by python.

Apart from that, implementing Machine4 encrypts was quite straightforward. As mentioned before, only 2 encrypts needed to be overridden as the rest of the methods would use virtual functions to encrypt with the most most recent encryptLetter.

Text

Description automatically generated

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Details | Data In | Expected | Actual Result |
| 4 Machine plugged phrase encryption | Encrypted phrase will be full “quick brown fox text” | Beta, IV, V, VIII, position 3,17,23,11 rings 8,9,3,4 plugboard "JK FR BE IP ZS AW", "QUICKBROWNFOX" | JFVKFEBWKTRTS | JFVKFEBWKTRTS |

A screenshot of a computer

Description automatically generated with medium confidence

With the encryption and simulation parts of the program complete, solving the enigma cipher would be relatively straightforward. The first step would to get the analysis functional. To do this, I started by making an abstract “Analyser” class, containing a pure virtual analyse function and a little bit of information about the Analyser being used.

Text

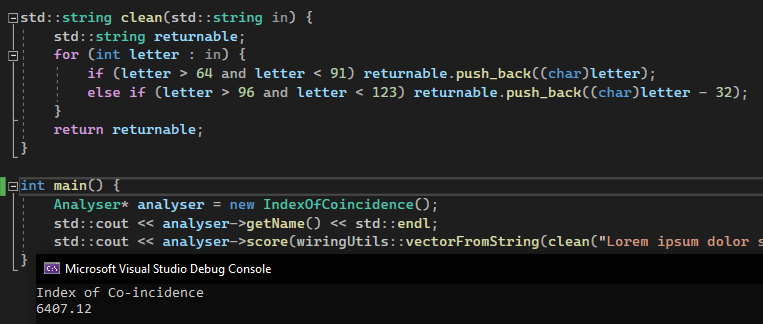
Description automatically generated

This basis would later allow me to contain a pointer to an Analyser in my SolverMachine, and call this score method to get a score of how close a solution was to correct during the attacks. Using this the index of coincidence was trivial to produce given the formula stated before.

A screenshot of a computer

Description automatically generated with medium confidence

This could be pretty easily optimised by removing the divisor part of the algorithm, and the scores would still be comparable as this divisor was the same for all inputs on a given ciphertext, however in the end this function didn’t end up taking much time during the attacks so the optimized version I produced alongside wasn’t too useful. Testing this with some lorem ipsum, this code would fail pretty badly despite its simplicity.



This output was far outside the expected, given that the paper claims that good text will be somewhere near 0.07 and bad around 0.04. Tracing this down (again with visual studio’s amazing debugging system) I realised that I shouldn’t have ignored the warnings that visual studio was giving me, as frequencies was being initialised with insanely negative values instead of defaulting to 0. All that needed to be done was to intialise the array.



Text

Description automatically generated

Next came a surprisingly complex and difficult part of development. Creating the N-Grams. From a previous attempt at this project, I already had a cleaned and well kept file of pure english, capitalised text. I began by creating a method would could produce N-Grams from a given input.

A screenshot of a computer

Description automatically generated with medium confidence

This allowed me to load up my pure english file, parse it as one long string and get all the NGrams before saving in my own format to be loaded at runtime. Before that, however, I had to decide how I would store and lookup the N-Gram frequencies. My original idea was to store the frequencies in a hash table (unordered\_map in C++) with the keys being an array of the letters to look up and the items obviously being the frequency. To store and load this map, I opted to use Boost Serialization, as from previous experience I know that manually coding file handling in C++ can easily become a great performance hog (especially with my level of skill) so I believed using another person’s library to do this would be beneficial. This came with its own host of problems, and required me install Boost and troubleshoot the CMake to include it. In this end however this came out to be quite simple.



This allowed me to create a dataBuilder file to parse the english text and write it to the file.

Text

Description automatically generated

This would quickly allow me to realise that my planned datastructure would be unneccessary and unusable. This problem being that C++ doesn’t know how to hash an array of integers. I could’ve written my own hash function to deal with this, however a friend quickly pointed out using a hashtable for this purpose would be unneccessary. Instead, an array could be built, with each index refering to a string of N characters and the value pointed to by this index being a frequency. For example, in the case of bigrams this would be used.

Text

Description automatically generated

Bigrams is a vector of bigrams in the big data set, so bigram refers to a single one of these. For each of these bigrams, the index would be made through bitshifts. I opted to use 5 bits per character as each char is between 1-26. This does leave a lot of empty space in the final array but even the quadgrams array is only around 4-8MB.

Despite changing the datastructure in use, I still decided to use Boost Serialisation due to its performance, simplicity and modularity. This left the databuilder code as follows. Soon after the screenshot was taken I introduced a log into the final arrays, producing a log table more akin to probabilities than frequencies, which should lead to better results.

Text

Description automatically generated

Visual studio gave warnings here as it didn’t understand the bitshift indexes, fearing that they’d go out of bounds. It also was unhappy about trigramFrequencies being allocated on the stack, but the code ran and produced the correct files so it’s nothing to worry about. Related is the usage of a pointer for quadGramfrequencies, as it would turn out to be far too large for the stack, causing crashes when ran. Instead it would be allocated on the heap by new()

With these files, I could construct the N-Gram analyser class(es). Due to the similarities between different types of N-Gram (with the only difference being in N), I intended to use a templated Class for this, only changing N where needed based on the template. This caused many linker errors.

Text

Description automatically generatedText

Description automatically generated

Even now, I am no closer to understanding why templates caused so many issues with the system structure. Even moving the method definitions to the header file couldn’t deal with these issues, as when they’d be imported to debug or binds the compiler would completely lose track of where the methods were/what their signatures were. To overcome this I had to distribute the NGrams over 3 different classes (bi, tri, quad) which may make the code less maintainable.

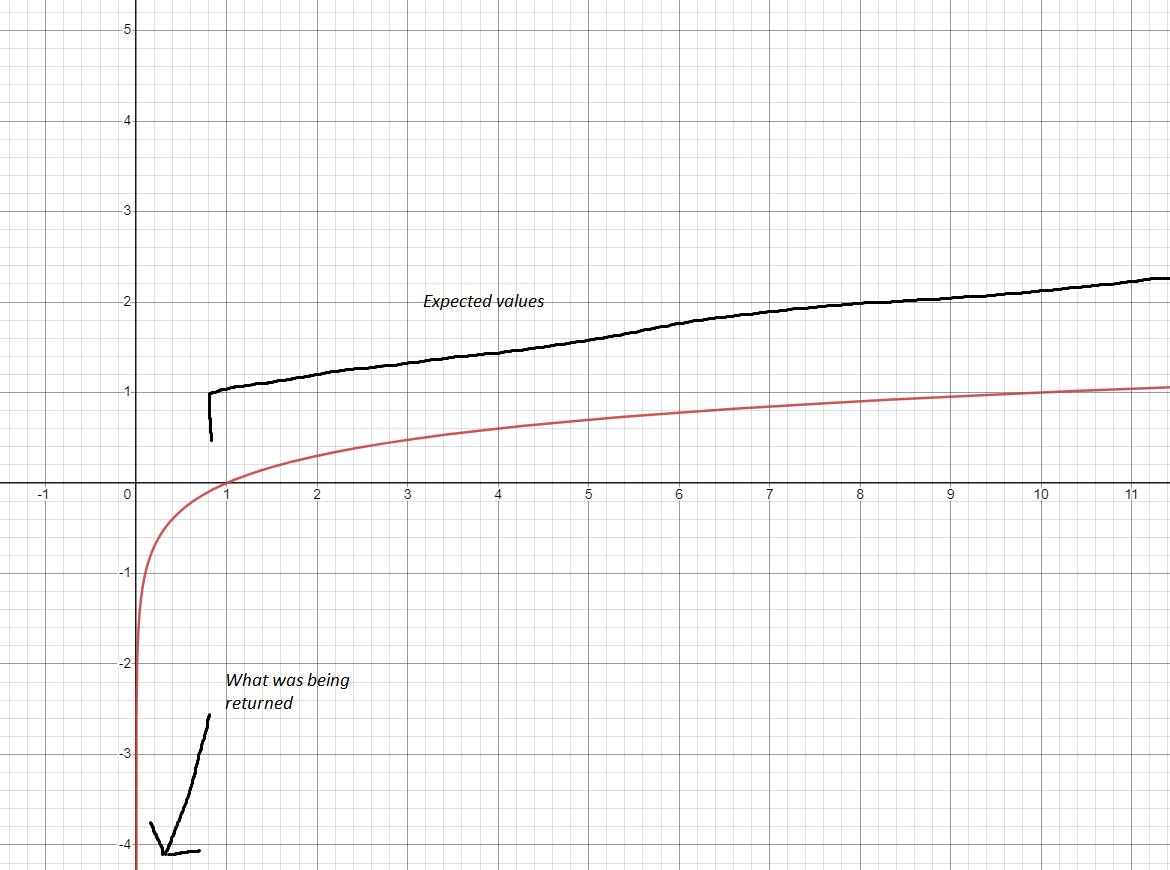
Text

Description automatically generatedText

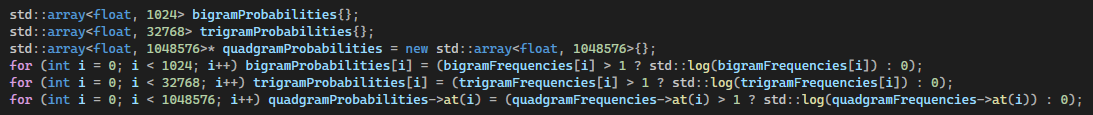
Description automatically generated

The initialiser takes in the filename of the related binary produced by the data builder, loads this with boost and puts it into the array. This is then scored by the score function, which takes in the word, converts it to a vector of NGrams then looks each bigram up in the array.

This originally was producing very small numbers with no difference between them, indicating a problem in the N-Gram code. Taking a look at the frequencies it was working with the problem became clear. Due to the fact that the databuilder was taking the logs of the frequencies, if any N-Grams weren’t found in the big data set, this would be taking the log of 0. Normally this is undefined and would crash, but C++ log desides to return a very small number (probably due to it’s estimation algorithm)



This would be quickly fixed by taking 0 for any NGrams not found in the data set, as opposed to taking the log (this is equivalent to finding 1 of the NGram)



To test these N-Grams, I decided to quickly implement a version of the final solver algorithm for the plugboard cracking step

Text

Description automatically generated

Sadly, this would crash, but not due to any bugs in the N-Gram or simulator code. The main issue came from the plugboard. As it was being updated, its reverse wirings wouldn’t be. To solve this I implemented a dirtying system, where the reverse wiring would be deleted and set to nullpointer after changing the plugboard in any way. However, this lead to strange deconstructor behaviour leading on from the original deconstructor design. To combat this, I finally got to use shared\_ptr and rewrote the wirings class to accommodate this.

Text

Description automatically generatedText

Description automatically generated

This solved the problems with the program, and allowed me to test how well the TriGrams (and thus N-Grams) solved the plugboard, with success up to 6 plugs on 400 characters in 150ms.

## (ii) TESTING TO INFORM DEVELOPMENT

*(a) Provide annotated evidence for testing at each stage justifying the reason for the test.*

*(b) Provide annotated evidence of any remedial actions taken justifying the decision made.*

# (4) EVALUATION (20 mARKS)

## (I) TESTING TO INFORM EVALUATION

*(a) Provide annotated evidence of testing the solution of robustness at the end of the development process.*

*(b) Provide annotated evidence of usability testing (user feedback).*

## (II) SUCCESS OF THE SOLUTION

*(a) Use the test evidence from the development and post development process to evaluate the solution against the success criteria from the analysis.*

## (III) DESCRIBE THE FINAL PRODUCT

*(a) Provide annotated evidence of the usability features from the design, commenting on their effectiveness.*

## (IV) MAINTENANCE AND DEVELOPMENT

(*a) Discuss the maintainability of the solution.*

*(b) Discuss potential further development of the solution.*

## APPENDIX A – BIBLIOGRAPHY