C++ Vector Field Plotter

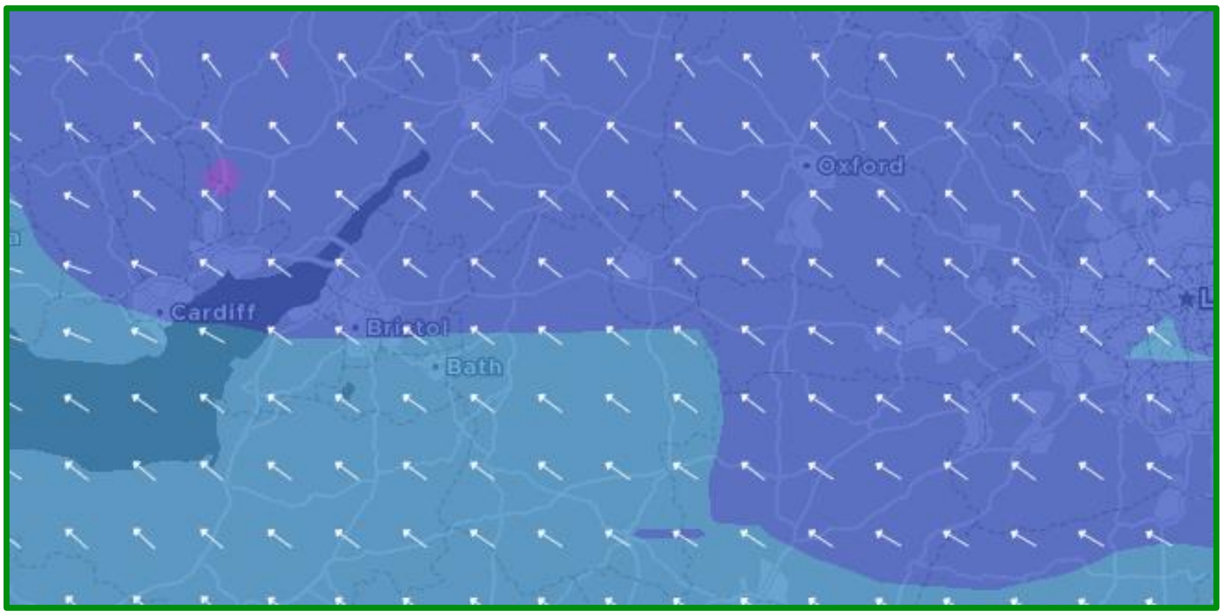
Harry Diviani

A-level computer science project

Analysis

Problem identification – a vector field graphing tool

Even though there are many examples of graphing software that can display the graphs of multiple mathematical functions simultaneously (for example Desmos, GeoGebra, or software on TI graphing calculators), there are not many applications that can display a vector field – a graph of a function that takes in a vector as an input and outputs another vector. Primarily, this application would be beneficial to students who are learning the topic in their maths class, but my solution would also provide an intuitive interface through which anyone can create and share art. Vector fields have various real-world applications such as modelling fluid flow or wind patterns.



This is a familiar example of the real-world applications of vector fields: you can use a vector function that takes in the location at some point on Earth and returns a vector describing the speed and direction of the wind at that location. We see this on the daily weather forecast, so clearly there are some important applications of this.

The solution will include an interface through which the user can use mathematical notation to input a function that takes in a 2D vector (x, y) and returns another 2D vector. A grid of particles will be displayed on the screen, and the velocity of each particle will be the vector function evaluated where (x, y) = the position of the particle. The particles will then move accordingly, and the movement of the particles gives a graphical representation of the vector function given by the user. Specifically, the particles would move along the “field lines” of the vector field. This solution requires a computer with hardware that is capable of displaying a graphical user interface.

Why the problem is suited a computational solution

This problem requires computational methods in order to find and implement a solution. Using a computer, it is possible to perform many calculations per second, which is needed for my solution, due to the hundreds of particles that must be updated and re-rendered.

Using a computer for rendering a vector field is more dynamic, meaning the user can quickly change features about the graph, such as the function being graphed or the graph's settings. A non-computational approach is unrealistic because, compared to a computational approach, manually graphing a vector field would be too time-consuming. Additionally, the notion of graphing 'moving particles' and creating art would not be possible.

A computational approach gives an accurate representation of what was entered (up to floating-point and numerical error), whereas manual, non-computational methods are subject to human error. If someone is using a vector field to model a real-life scenario - wind patterns, for example - the vector field should be completely accurate. This can be accomplished by using software that displays the vector field, which is what my solution will aim to accomplish.

If a teacher or student creates a graph with the program, they may wish to share it with the rest of the class. The Internet has made it significantly quicker and easier to share content with the rest of the world. My solution would include a feature in which the user can save graphs they've made onto a database, as well as find graphs others have made that are saved onto the database. This is one of the reasons the problem lends itself to a computational solution; users would be able to share and find graphs others have made through the Internet. For this I will use a MySQL database on an external server.

Stakeholders

My maths teacher Mr. Mockford has agreed to be a stakeholder for my project; I think it is relevant to him because he is a teacher with a maths degree, so not only would he be able to use the program in a meaningful way (to teach kids) but he would be able to see whether or not the program works correctly (i.e. gets the right results).

I have also asked my physics teacher Mr. Davis to be a stakeholder, for similar reasons. Physicists often use vector fields in their work.

Both of my stakeholders are teachers; I have chosen this because the purpose of the program is to help with education, to help students visualise what is going on with a vector field equation. I think they will also be able to help me & give me guidance on the educational aspect of the program.

Computational methods that can be used

Problem recognition:

The overall problem I am trying to solve is to create a program that anyone can use to create and view a vector field. The main, underlying problem will be calculating and rendering the movement of hundreds of particles, 60 times per second. This falls into the category of 'applying a mathematical function to multiple objects', so the program could be optimised to take advantage of SIMD and GPU parallel processing. However, this program should be able to run on school computers, which typically do not have a dedicated GPU. One large problem will be optimising my program to the point where a 60FPS graph of a vector function can be viewed on underpowered hardware.

Another problem will be creating a reliable database structure on a server that can be accessed safely (i.e., no SQL injections are possible in the program) and can serve requests from a number of clients that are searching for graphs on the database.

Decomposition:

The problem described can be decomposed into a number of smaller, more manageable sub-problems. In this document, the overall problem will be solved through the principle of divide and conquer.

This is an initial idea of what the steps are to solve the problem of creating the vector field:

1. A grid of particles (dots on the screen) is displayed.
2. The user inputs a vector function, q(x, y) = <f(x, y), g(x, y)>. This q(x, y) determines the velocity of a particle with position (x, y) in the main window.
3. An algorithm parses the user input into an expression that can be understood by other functions in the program. The intricacies are described in the design section.
4. The velocity of every particle on the screen is updated according to q(x, y).
5. The particles move for 1/60th of a second (because 60FPS)
6. The particles are re-drawn, and the display is updated.

The problem with step 5 is that we can't set a fixed timestep of 1/60th of a second for the particles to move, since it's unlikely that the framerate will reach 60 on underpowered PCs. If the framerate of the program fluctuates while the timestep stays constant, then the speed of the particles would appear to change and so the program may give a less accurate graph.

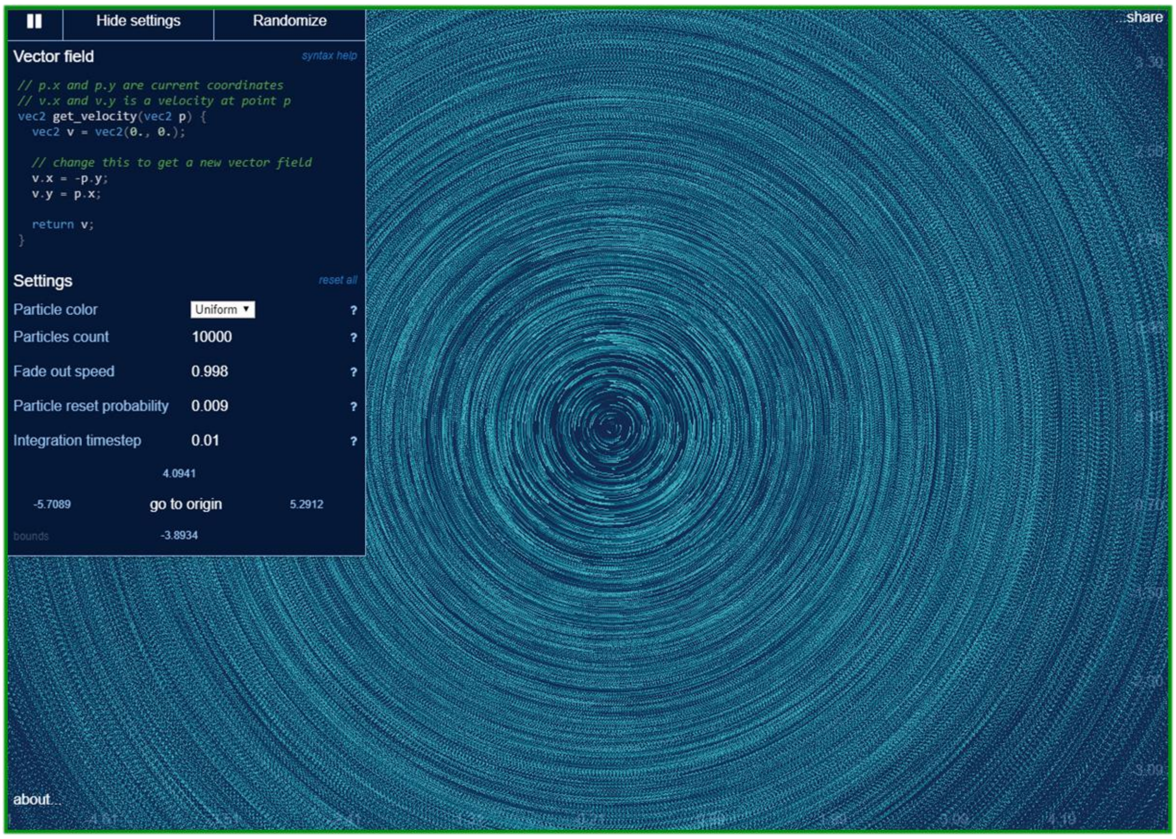
This is significant since the target audience for such an application would be students/teachers and so it would likely be run on school PCs, which are notoriously underpowered. I looked at how to solve this problem in the design section.

In my research I will look into alternative methods for updating the position of the particles to further ensure that the program can be run at a good framerate on underpowered machines. Specifically, I will look into various computational methods of solving the differential equations that relate the position and the velocity of each particle, since this will likely be the most difficult thing to optimise but can save a lot of computation time.

Research

Existing Similar Solution – Fieldplay

Fieldplay is an open-source web application that can run on any device and browser that supports WebGL. It allows the user to enter a vector field through GLSL (OpenGL Shading Language); a high-level language with syntax based on C. Code written with GLSL is executed directly on the GPU.



Vector field

```
// p.x and p.y are current coordinates
// v.x and v.y is a velocity at point p
vec2 get_velocity(vec2 p) {
  vec2 v = vec2(0., 0.);

  // change this to get a new vector field
  v.x = -p.y;
  v.y = p.x;

  return v;
}
```

Settings

Particle color	Uniform
Particles count	10000
Fade out speed	0.998
Particle reset probability	0.009
Integration timestep	0.01

bounds: 4.0941 / -5.7089 go to origin 5.2912 / -3.8934

The user interface is simple, with an area for the user to enter code, and list of settings the user can adjust to their liking which can change the appearance of the vector field. The vector field itself takes up the majority of the display, and the user can hide the settings panel at any time.

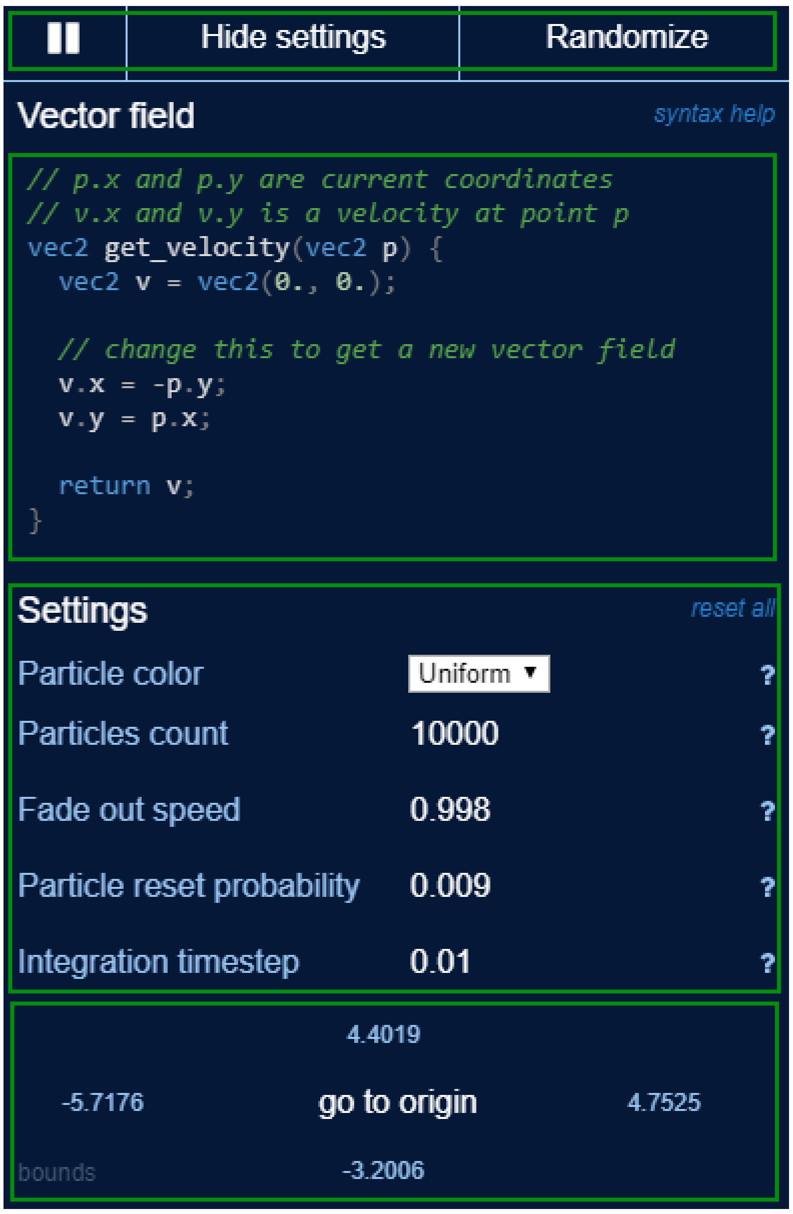
This program is similar to my idea for a solution, in that it displays the movement of particles on the screen according to the position of those particles. I have found the application useful for plotting 2D vector fields. However, it is not intuitive and not all users would want to enter code to plot their vector field, which is why my solution will use math notation instead. This will make it easier for the user.

The pause button stops the movement of particles on the screen.

The user enters a vector field by specifying a section of code to execute on the GPU. v.x, v.y denote the x and y components of a particle's velocity.

Here the user can adjust the settings of their graph. There are 3 settings for particle colour which change the colour of a particle based on its speed and direction.

The user can zoom in and out of the graph by changing the scale on the x and y axes.



Vector field

```
// p.x and p.y are current coordinates
// v.x and v.y is a velocity at point p
vec2 get_velocity(vec2 p) {
  vec2 v = vec2(0., 0.);

  // change this to get a new vector field
  v.x = -p.y;
  v.y = p.x;

  return v;
}
```

Settings

Particle color	Uniform
Particles count	10000
Fade out speed	0.998
Particle reset probability	0.009
Integration timestep	0.01

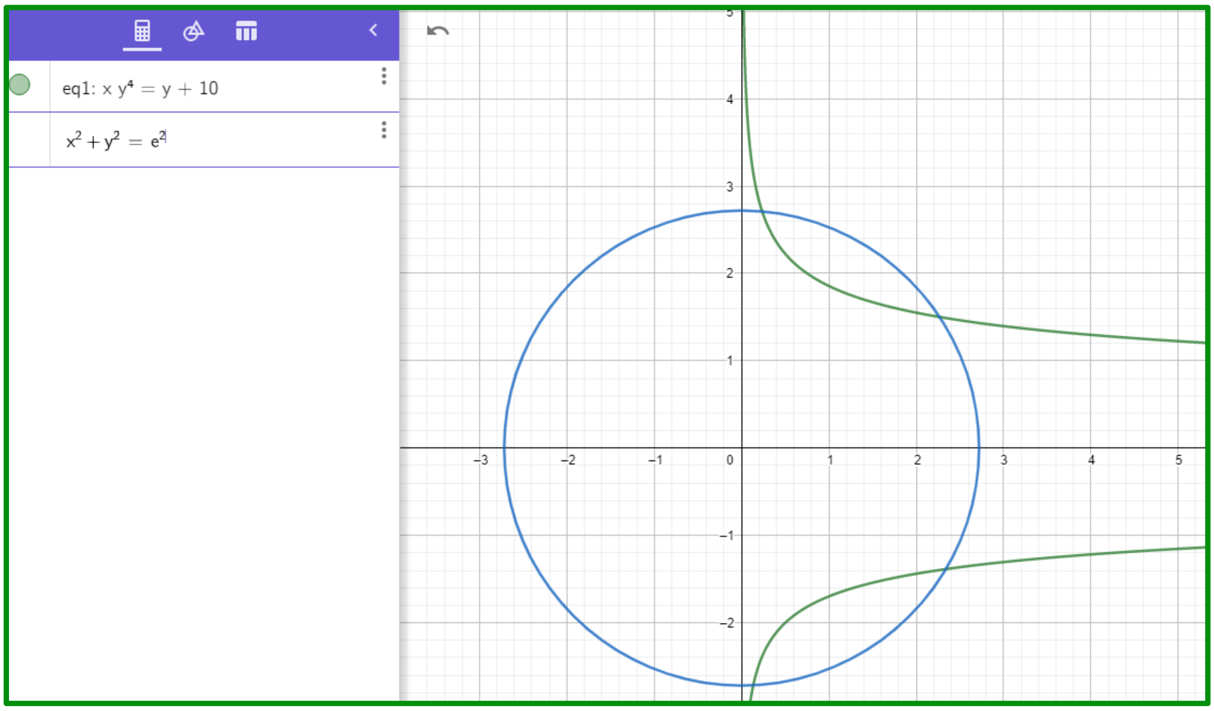
4.4019
-5.7176 4.7525
-3.2006

Parts that I can apply to my solution:

Fieldplay is very versatile in that the user can define their vector field through an algorithm and not just a mathematical function. This makes it highly customisable but also unintuitive; I believe that most people would prefer to enter a vector field with mathematical notation instead of GLSL code because vector fields are taught as a concept in maths. The takeaway from this is that it may be better to limit the user's input to a mathematical function rather than a section of code, as this will also make it more familiar to students and teachers who are using the program in class.

In my solution, we are met with the problem of finding the position after 1/60th of a second of a particle given its velocity and current position. We could add the particles velocity over 60 to the particle's current position using displacement = velocity * time. However, this would be inaccurate; the particles would only move completely accurately if there were an infinitely small timestep. To make the approximation more accurate, the developer of Fieldplay has implemented a higher-order Runge-Kutta numerical method for solving the differential equation that links the particle's velocity to its position. What this means is that, after the user has entered their code, the Runge-Kutta algorithm is executed in order to find the function that describes the position of a particle at time t given their initial position. I will research methods for doing a similar thing for my solution.

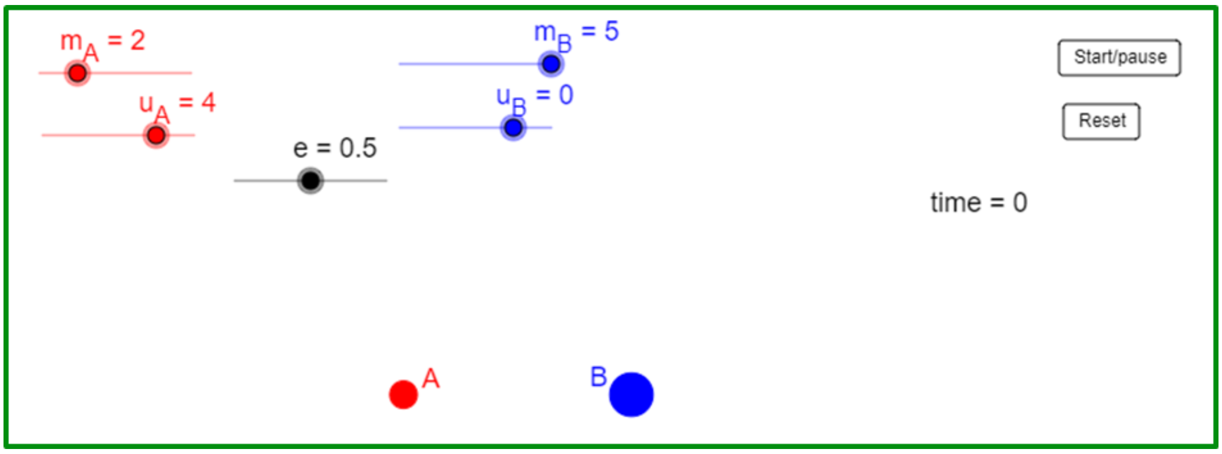
Existing Similar Solution – GeoGebra


eq1: $x\,y^4 = y + 10$

$x^2 + y^2 = e^2$

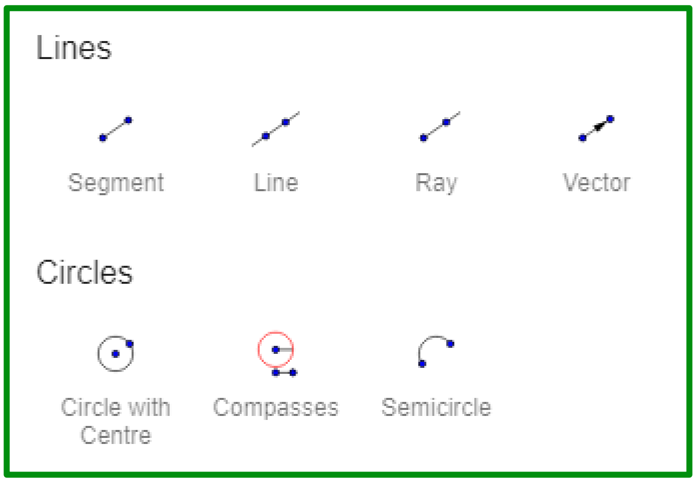
GeoGebra is a web application that acts as a 2D or 3D graphing calculator. The user can enter multiple functions simultaneously and see where the curves intersect. This graphing calculator is most useful in schools when learning algebra or calculus.

GeoGebra also acts as a platform on which teachers can create interactive resources for students. For example, Edexcel includes links to GeoGebra resources in their textbooks.



This interactive resource by Edexcel illustrates collisions between two particles. This shows how versatile GeoGebra is and how it can be used to aid in student's learning.

Another notable feature is that the users does not have to enter the equation of a circle or line to plot it on the graph. Instead, there is a menu with geometric shapes. This makes it easier and quicker to work with overall.



Overview: Even though GeoGebra was not made for vector fields, it still shows some useful features which I can incorporate into my application.

Existing Similar Solution – Desmos

Desmos is another web application which is similar to GeoGebra in that it can plot 2D curves. One useful feature about Desmos is that it allows you to enter the function you would like to plot with mathematical notation (as opposed to plaintext like f(x) = 3x^2), for example:

$$f(x) = \int_0^x (t + 10)\, dt$$

This is especially useful for the user because it allows them to verify that they have correctly entered the function they want to plot. I will try to incorporate this into my application.

Success criteria

In conclusion to my research, I have made success criteria to meet for my solution.

-A main menu with buttons: This is for user interface purposes, switching between parts of the program.

-A list to browse vector fields from a database: “List” for user interface purposes; “vector fields in a database” for usability purposes (i.e., coming back to a vector field you made earlier or loading a vector field someone else made)

-A plotting screen: Essential feature as described earlier in the analysis; there needs to be a plotting screen so that the vector field can be plot.

-User can input vector function: Essential feature, user needs to be able to input the function so that the program knows which function to plot.

-Program takes in user keyboard input for vector function: Other types of input (voice? Camera? Handwriting recognition?) I believe are not worth the time and effort to go into when you consider how typical graphing software like Desmos are used, there’s handwriting recognition software like Photomath, but this is not a mobile application.

-A grid with particles: Essential feature. By “grid” I mean a 2D integer lattice i.e., points inside of a desktop window. Particles will be created on the grid & this is a feature intrinsic to how the solution will work from a technical standpoint.

-The particles move according to the vector function: Essential, this is simply how the solution will work mathematically

-The ability to save the vector field to a database: Usability feature, for sharing with other people or coming back to it later.

-Ability to go back to main menu: Basic UI feature.

-Ability to configure settings on the vector field plot: Usability feature for example increasing the speed of the plot to see long term behaviour of a field or changing particle colour for aesthetic purposes.

Limitations

There are some limitations with the proposed solution.

The first limitation is a technical one; it turns out that numerical methods for solving differential equations are computationally intensive and so it will be difficult to display a large number of particles (since a separate differential equation needs to be solved for each one). Hence it will be a large problem trying to optimise the numerical integrator to solve this, perhaps by multithreading, we’ll see.

The second limitation is where the solution makes the program less usable. The program will only work for 2D vector fields. However, 3D vector fields exist, and are actually a lot more important with regards to applications in physics and stuff like that. If I had the knowledge to do 3D rendering, I probably would have gone this route instead. 3D rendering is much more complex than 2D rendering and in order for this project to be completed before the deadline I have chosen to not undertake the monumental task of learning how to do all that.

With that said, in classroom environments 2D vector fields are usually taught first in classroom environments as they are simple to draw on a whiteboard and the principles are mostly the same as 3D vector fields. Hence there is still pedagogical value in an application that displays 2D vector fields over 3D ones. In fact, this may be a non-problem—people who would benefit from a 3D vector field plotter likely already know how to use them (engineering & physics students learn MATLAB, R, etc.) and there would not be much point to them learning something new for no increased functionality, especially a program which is so specialised for students.

A third limitation is one in the context for where the program will be used. I have said that the program will be made to be used in classrooms. If this is the case, and it will be used in maths classrooms, it might not be likely that the students will have immediate access to a computer. It may have been a better idea therefore to create a mobile application rather than a desktop application, so that students would be able to use the app on their phone while in lesson. I justify the choice to create a desktop application similar to how I justify using a 2D vector field instead of a 3D one—I do not know how to make iOS or Android apps and I do not believe it would be possible to learn from scratch and still get the project finished before the deadline. Furthermore, when I began this project, I did not have access to a device with iOS nor Android so it would not have been possible to test it regardless.

In my defence, just because students can't use the program in class doesn't mean they can't use it at all. In order to help with what they are learning in class they can use it at home so they can easier visualise what is going on. And as I mentioned earlier teachers can upload their vector field to a database for students to look at later. So, I don't really see this as a big limitation or a fatal flaw in how the program applies to the context of the situation.

The fourth limitation is in the context of how the program is presented. As it stands the program is a desktop application—if this were a real product this means the user would have to go on a website and download a release for the application before opening it. But nowadays we have applications like Desmos and GeoGebra and even Fieldplay which do everything in the browser. It is a bit messy to have these graphing calculators which work in the browser while, in contrast, there is a desktop application with an extremely specific purpose which you need to download and run. I believe this alone would deter most people from using the product, simply because everything else just runs in the browser. There is also an actual, tangible issue with it needing to be installed beforehand—you typically can't install things on a student account on a school PC (though maybe things are different in university); therefore, if it is to be run on school PCs the IT admin would have to install the application manually—it would be hard to convince them to do this, so to run it the program would need to be executable without installation (which would require the user to download a pre-compiled executable file) but of course executable files contain machine code which are processor-specific so this is not really possible.

Looking at both sides, let's assume I chose to make the application a webapp instead of a desktop app. Then, in order to maximise performance, I would have to learn how to use all of the funky shiny web technologies created by some large tech companies which, again, there is not much time, and sometimes I wonder if it's a good idea to make an entire project based off of the code of a large private corporation like that. Yes, the libraries might be open source, but now GitHub is owned by Microsoft. Anyway, if it were a webapp it would most likely not run as quickly due to JavaScript being an interpreted language whereas C++ is compiled. Furthermore, the C++ implementation has less layers of abstraction over the graphics interface (SFML -> OpenGL -> bare metal) whereas JavaScript would have some unruly amount. That is the primary reason why I choose C++ over JS. I recognise that the developer of Fieldplay made it work but really there is not much time. I have a personal distaste when it comes to using JS to make fully fledged applications like this which I won't get into here. But, in short, I think the longevity of the application will be a lot longer if I code in C++ than if I implement the solution as a JS webapp.

Essential features

There are essential features which, if not implemented, make the whole program pointless because without them the program would not do what it is made to do.

-The program needs to be able to take a user input for a vector function. This is essential because the program needs to know which vector function is to be plot before the program can actually plot it. I have chosen that the user will input the vector function manually with their keyboard or load it from a database for usability purposes.

-The program needs to be able to parse, correctly, this user input (from the keyboard) into a readable format which is understood locally by other parts of the program. This is essential because the program needs to be able to use the user's vector function in order to plot the vector field. And it needs to be parsed correctly, otherwise the program might plot a completely different function to what was input and get a completely wrong result. I look at this in the design section.

-The program needs a GUI with a 2D rendering window which displays the plot. This is essential simply because the plot is a visual plot of a 2D vector field and so it will be rendered on a 2D window. It is not reasonable to render the particles in a CLI, although I'm sure it's possible, the problem simply lends itself to using a GUI with a 2D rendering window. This render window will display the plot of the vector field (the particles moving around.) There needs to be a render window because of course the whole point of plotting something is so that the user can see it on their screen.

-The program needs to produce an accurate plot of the vector field. This is essential because there would be no point in using a tool if it gets you the wrong answer. Imagine a calculator that returns 3 if you enter 1+1. There would be no point. As stated, earlier if the particles move with a velocity determined by the vector function, then they will move along the field lines of the vector field. We can make a fairly accurate (near perfect) plot by using a numerical method with a high enough accuracy as discussed in the design section.

Software & hardware requirements

Due to these essential features, there are hardware and software requirements for the machine running the program.

The machine needs to run Windows, specifically Windows XP and up. This is partly because the graphics library I am using (SFML) only works on XP and up if it is running on Windows. The reason why Windows itself is needed is because I am compiling the application using Visual Studio, which produces an .exe. However, Linux and MacOS users might be able to use Wine (a compatibility layer for Windows) in order to run the program – in this vein I will get some people I know who use these OSes to test it on their machine. SFML is only 2 layers of abstraction over the bare metal so I think it will work on many different types of PC.

They will also need Visual C++ runtime and the SFML libraries. When I need people to use the software for testing, I will give them an installer which installs the dependencies alongside the application itself.

With regards to hardware, there needs to be some sort of graphics processor on the motherboard. There doesn’t have to be a dedicated graphics card but there at least needs to be an integrated graphics processor in the CPU. This is because the program will use shaders, which are executed directly on the graphics processor. This is just for aesthetic purposes and this will get scrapped if I find out it doesn’t work on school computers.

The PC also needs a CPU which has the ability to run the numerical integrator described in the design section with a reasonable computation time. The user will be able to change the number of particles & the integration step size to reduce or increase the computational load. I will test the program on school computers to benchmark how everything will go.

The PC needs a keyboard, mouse and monitor for obvious purposes.

Design

Potential methods to be used

The overall problem for the program can be decomposed into multiple distinct subproblems, each one being easier than the overall problem. Then the overall problem will be solved by tying everything together. There are 4 large problems which will be addressed and solved one-by-one in this subsection.

The first subproblem is the problem of solving the differential equation given by the user so that the particles actually move on the screen in the correct direction, so as to trace the field lines of the vector field. This is fairly complicated and there are many different algorithms that have been invented to solve this subproblem. These algorithms are called numerical integrators. I will need to compare some of these algorithms in order to choose the most suitable one for the context of problem, mainly considering the limitations of school PC hardware.

The second subproblem is the problem of parsing the users input. The user will input their vector function using a keyboard. The problem here is that it isn't obvious how the text will be parsed, so I will have to look into methods for parsing the input into something that can be understood by other parts of the program.

The third subproblem is with regards to the structure of the program itself. I need to come up with a way to structure the program that will make the codebase organised, while making sure that the program is performant and functional. The codebase needs to be organised so that it is easy to make changes and come back to old code. By organised this means the program will be modular and have commented sections of code. This points towards using OOP as a suitable paradigm for the solution.

The fourth subproblem is with regards to the context of where the program will be used. The types of computer this program will be running on will likely not be that powerful, so I need to implement an algorithm that ensures the program still works correctly despite it not running at a high framerate. This is an important problem because we need to make sure that the program is still functional on school computers, simply because the program is made to be used in an educational context.

Numerical integration algorithms

In this subsection I look at different numerical algorithms for solving differential equations, going into them, and weighing pros and cons in hopes of choosing the best one for my program.

The differential equation at hand is:

$$\begin{pmatrix} \mathrm{d}x/\mathrm{d}t \\ \mathrm{d}y/\mathrm{d}t \end{pmatrix} = \begin{pmatrix} f(x,y) \\ g(x,y) \end{pmatrix}$$

with the initial condition of (x(0), y(0)) = (x_0, y_0). (x_0, y_0) is the point on the grid where the particle “spawns” and will be determined inside of the program. (x, y) is then the position of a particle at time t. This is a first-order, ordinary, coupled, autonomous differential equation.

Since the functional form of f and g are unknown beforehand, as they will be input by the user, there is no way of writing an analytical solution for x and y in terms of t, as you would be able to if f and g were known beforehand (e.g., if f and g were ax+by and cx+dy). The only way for the program can account for all functions f and g is by using a numerical method, which only uses the values of f and g at (x, y) and does not depend on their analytical properties such as their derivative.

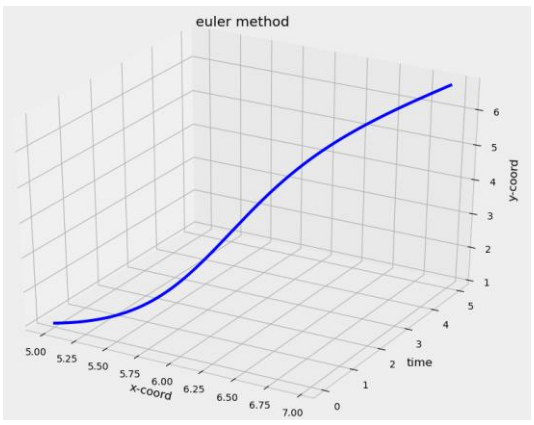
Euler Method

The Euler method is a simple method to solve a first order ODE given initial conditions. Let r = (x,y) for shorthand so that the differential equation at the top of this page can be written as r’ = q(r), where q(x,y) = (f(x,y), g(x,y)). The goal is to solve for r at a given time t. The method is as follows; we know the initial position r(0) = (x_0, y_0) of the particle and the initial gradient q(r(0)). The gradient represents the direction of the particle. Therefore, by moving in this direction for a time h you can get an approximate value for the position at time t=h, then you can repeat the process to get an approximate position at t=2h, 3h, … until you get to the value of t you want. So, the solution is given by r((k+1)h) = r(kh) + h * q(r(kh)).

On the next page is an implementation of the Euler method in Python. It is easy to implement but it computationally not as accurate as some other methods. I have used an example of q(x,y) = (x/y^2, x-y). This is just a basic implementation for illustration purposes, it’s not actual code that will be used in my project.

Due to the single for loop, which has n=T/h iterations, the time complexity of the Euler method is O(n) = O(1/h). So, if the step size h increases by a factor of 10, then the time taken to run the algorithm decreases by a factor of 10. However, the method is a first-order method, meaning the error of at a given time is O(h); so, if h increases by a factor of 10, then the error also increases by a factor of 10. In fact, if h is too large the Euler method will actually look nothing like the actual solution & will diverge. And this limitation just gets more demanding for more complicated functions, so-called stiff equations. So, there is a trade-off between accuracy and computation time. If h is too large the approximated solution won’t be accurate; if h is too small the algorithm will take too long to run. This is especially important when considering that the program is made to be run on school computers.

```
import numpy as np #vectors
import matplotlib.pyplot as plt #plotting
from mpl_toolkits import mplot3d #3d stuff
def q(r: np.array) -> np.array:
  return np.array([r[0]/(r[1]**2), r[0]-r[1]]) # example
n = 50000 #use 50000 steps
T = 5 #go from t=0 to t=5
h = T/n #50000 steps from t=0 to t=5, so each step has a time diff.
of 5/50000
r = {0: np.array([5, 1])} #one way of implementing; key is time, val
ue is position at time
for k in range(n):
  r[h*(k+1)] = r[k*h] + h * q(r[k*h])
fig = plt.figure()
ax = plt.axes(projection="3d")
ax.set_title("euler method")
ax.set_xlabel("x-coord")
ax.set_ylabel("time")
ax.set_zlabel("y-coord")
timeline=np.linspace(0, T, n+1)
xline = np.array([pos[0] for pos in r.values()])
yline = np.array([pos[1] for pos in r.values()])
ax.plot3D(xline, timeline, yline, 'blue', linewidth=3)
plt.show()
```



The fact that the Euler method need a very small h to be numerically stable for some functions q(x,y), paired with the facts that the time complexity is O(1/h) and user can put in an arbitrarily complicated function, leads me to the conclusion that the Euler method isn't a reasonable choice for the program.

4th order Runge-Kutta

The method I think will work best is the RK4 method. It is like a more sophisticated version of Euler's method. It uses 4 different directions at different points instead of just using the one. Anyway, its error is $O(h^4)$ compared to the O(h) error of the Euler method. So, if h decreases by a factor of 10, then the error decreases by a factor of 10,000. It has the same time complexity as Euler's method; O(1/h). Furthermore, it is more numerically stable, so I can't really see any reason to not use this over Euler's method. This is the same method that was used in the Fieldplay application that I looked at in the analysis section.

```
import numpy as np #vectors
import matplotlib.pyplot as plt #plotting
from mpl_toolkits import mplot3d #3d stuff

def q(r: np.array) -> np.array:
  return np.array([r[0]/(r[1]**2), r[0]-r[1]]) # example

n = 50000 #use 50000 steps
T = 5 #go from t=0 to t=5
h = T/n #50000 steps from t=0 to t=5, so each step has a time diff.
of 5/50000
r = {0: np.array([5, 1])} #one way of implementing; key is time, val
ue is position at time

for k in range(n):
  k1 = q(r[k*h])
  k2 = q(r[k*h] + h*k1/2)
  k3 = q(r[k*h] + h*k2/2)
  k4 = q(r[k*h] + h*k3)

  r[h*(k+1)] = r[k*h] + h/6 * (k1 + 2*k2 + 2*k3 + k4)
```

Code not actually going to be used in my project, it's just for showing how the algorithm could be implemented

Math parsing

In the program, the user will have to type their vector function they want to use into an input. This will be composed of 2 separate input textboxes: one for f and one for g. Each box will accept a single mathematical expression which may contain x and y. This expression represents the function.

There are 3 common ways in which an expression can be typed out. They are postfix notation (aka. reverse Polish notation), prefix notation (normal Polish notation) and infix notation.

The Polish notations are easier to parse and evaluate. However, the input will be coming from the user, so, in order to make things as intuitive as possible, the program should accept the type of input which the user will be familiar with (infix notation). So, the program will accept infix notation and not any of the other two.

In this subsection I figure out how the input will be parsed.

Shunting-yard algorithm

Postfix notation is very easy to evaluate using a stack. For example, the expression 3 5 * 2 + (which is 3*5 + 2 in infix) can be evaluated like this:

Step	Expression	What happened	Stack contents
0	3 5 * 2 +	Nothing	
1	3 5 * 2 +	Read L->R until operator	3, 5
2	15 2 +	Apply * to the top 2 stack items	
3	15 2 +	Read L->R until operator	15, 2
4	17	Apply + to top 2 stack items	

(The rightmost element of the stack is at the top of the stack)

So, it will be easy to evaluate the expression if it is first converted from infix to postfix. To do this a stack & queue are used in tandem. Here is 3*x + 2*y being converted to postfix notation. This is the shunting-yard algorithm. Numbers are pushed immediately to the queue; operators are pushed onto a stack & popped off according to BIDMAS rules or at the end of the expression.

Step	Expression	Output queue	Stack content	What happened
0	3*x + 2*y			Nothing
1	3*x + 2*y	3		Numeric queued
2	3*x + 2*y	3	*	Operator pushed
3	3*x + 2*y	3 x		Numeric queued
4	3*x + 2*y	3 x *	+	* precedes +
5	3*x + 2*y	3 x * 2	+	Numeric queued
6	3*x + 2*y	3 x * 2	+, *	Operator pushed
7	3*x + 2*y	3 x * 2 y	+, *	Numeric queued
8	3*x + 2*y	3 x * 2 y * +		End operator pop

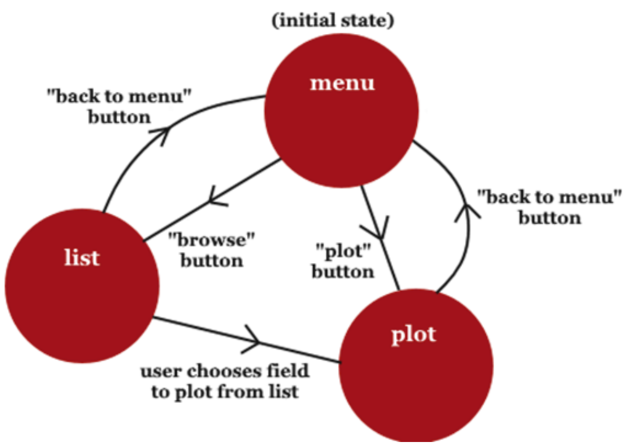
Then the postfix evaluation can be used as needed as the RK4 algorithm is carried out. I think this will work well for the program. I believe the choice to use the shunting-yard algorithm is justified, simply due to the nature of how expression will be input.

State machine implementation

There are three states the program can be in:

1. *menu,* when the program is in the main menu,
2. *plot,* when the user is plotting a vector function,
3. *list,* when the user is looking at the list of vector fields on the MySQL database.

State diagram:



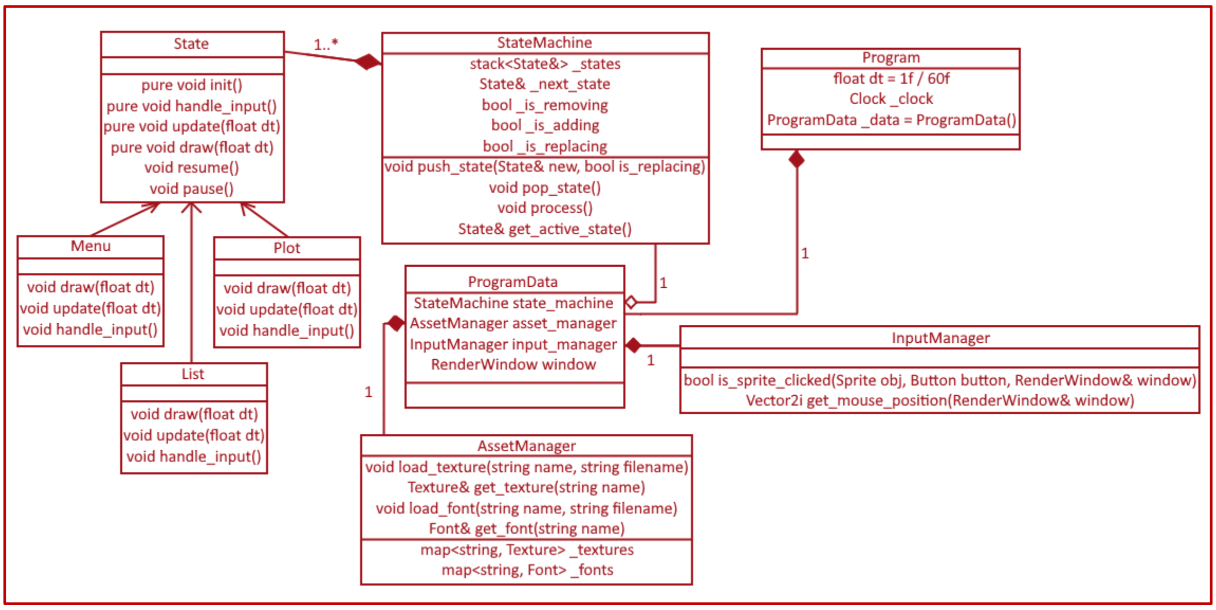
It would simplify the coding of the program & make everything a lot easier overall if I were to treat these states as separate but related entities. The menu, list and plot states are similar in that they're states, but they have different logic & data used in their execution. Therefore, it seems reasonable for there to be a parent State class, which will have a set of primitive methods such as pause() (to pause execution of the state), resume(), etc. paired with a set of pure virtual methods: update(), which will dictate what occurs in each frame, draw(), for rendering on the screen, and handle_input() to handle input. These pure virtual methods need to be defined by a derived class, and so they will be what differentiates the states from each other. Each of the 3 states will be implemented as a subclass of the State class and will provide their own implementation for the pure virtual methods.

Therefore, I will definitely be using OOP in the coding for the solution. This way of structuring the program makes the most sense with regards to the program flow & how the user will be using the program.

There has to be a logic in the program that controls the flow of states, the states cannot exist in a vacuum if the program is to work as intended. There will be an entity that controls the states. To do this there will be a stack structure which has the current state at the top of the stack. When a state transition occurs, the current state will be paused, and the new state will be pushed onto the stack & the entity will initialise and resume the new state. If the new state is to replace the current state, then the current state will be popped off beforehand. If the current state is finished or we need to go back to the previous state, then the current state can just be popped off & now the current state can be resumed. This is why a stack structure is useful.

Here is a diagram illustrating the different classes that will be used in the program.

UML Class Diagram



I have also included AssetManager and InputManager classes which exist for UI purposes. The AssetManager controls the fonts and the textures (e.g. images for buttons on the screen). The InputManager class is used to check if a sprite is being clicked on, which will be used for button clicks & the like.

The listed properties for the Menu, Plot and List state are empty but they will have properties of their own which will be used. For example, the Plot state will have its own internal clock which will hold the time t as described earlier in this section.

Texture, RenderWindow, Vector2i, Font, Clock, Sprite and Button are classes from SFML. This is the library which I have chosen for rendering 2D graphics. It is an abstraction over OpenGL and has a more intuitive API which will make coding a lot quicker. It does not have much overhead so it will still be suitable for school computers.

Framerate independence

Update and draw methods take a float dt as a parameter; this is done to allow for framerate-independent playback. The program is made to run at a constant 60FPS. However, some computers won't have a powerful enough processor to achieve this and so the FPS will be lower. Without framerate-independent playback the particles on the screen will actually move slower in real-time than if there were framerate-independence, since the time in the program would be based on the number of frames completed instead of real time. This is fixed by recording how long it takes (real-life time) between frames and comparing it with the ideal framerate of 60FPS to correct the velocities of the particles to what they should be.

There will be an internal clock inside the program (more specifically, the Plot state) which starts at t=0 and moves at a rate set by the user. This is the t which is used in the differential equation. For example, let's say the internal clock moves at a rate of 3 seconds per real life second. If we were running at a smooth 60FPS then the time between frames is 1/60 seconds. So, each frame we can increase the internal clock by 3/60 = 0.05 seconds. Now lets say that we aren't running at a smooth 60FPS but instead there's a random time dt between two adjacent frames. If dt is twice the change in time there would have been at a smooth 60FPS then we should increase t by twice as much as we would have because twice as much time has passed in real life. In general, if dt is k times the change in time for 60FPS then dt =

k * (1/60) so k = 60dt. We should increase t by k times as much as we would have, so the change in the internal clock time between two frames will be (3/60) * k = (3/60) * 60dt = 3dt.

If the user sets a clock rate of R seconds per real life second, then the internal clock increase per frame will be R * dt.

Tying it together

I have established which algorithms I will be using, which solves each subproblem. Now I need to tie everything together to make it clear what the overall design of the program will look like.

When the user opens the program, a main menu state will be created as described in the state machine subsection. This main menu state will have buttons saying Plot, List, Exit. If they will click a button that says "Plot", the plot state will be pushed onto the stack. They will be greeted with two input textboxes where they input the vector function that they want to plot, as well as a slider which dictates how fast the simulation runs compared to real-time. After clicking a button which says "go", the function input is then parsed using the shunting-yard algorithm described earlier. The function will be stored internally as a postfix sequence of operators and numeric values which can be evaluated easily using a stack structure.

After the input has been parsed, an internal clock begins at t=0. This represents the value of t in the differential equation described earlier and will be stored in the data for the Plot state. The rate at which t increases will be dictated by the slider as mentioned in the previous paragraph. As justified in the framerate independence part of the last page t will increase by R * dt each frame where dt is the time since last frame and R is the value on the slider.

The 4th order Runge-Kutta algorithm will be invoked using the internally stored postfix representation of the function entered by the user paired with the value of t mentioned in the last paragraph. This will be used to generate an array of maps, which is held in the Plot state data, where each map relates each time t to its corresponding x-y coordinate for the particle, and each particle will have 1 map in the array.

Usability

One usability feature that is included in the design is the use of a keyboard to enter the vector function. I have chosen this over other methods because it is likely that the user of the application will be familiar with other software, such as Desmos, GeoGebra, etc. which have you enter equations in the same way. And anyway, there aren't really many alternatives. Two alternatives I can think of are: Using a camera & handwriting recognition, speech recognition, both of which are obviously not suitable, since if the user doesn't have a camera they can't take a picture of the equation, and if they have no microphone they PC can't pick up on what they're saying. But in contrast 100% of users who can run the program will have a keyboard since you need a keyboard plugged in to boot your PC.

Another usability feature is that I have designed the program as a GUI rather than a CLI. I need this to be the case simply due to the nature of how the vector field will be plot, and the fact that things are being plot at all. Imagine Desmos or GeoGebra in a CLI. It would not be reasonable, because the large size of characters in a CLI means that you wouldn't be able to view intricacies in a complicated graph. And this is even more limiting in my program because we have a large number of particles on the screen at the same time and I don't really see how you would be able to discern what's going on in a CLI implementation of this problem. I have tried to justify it but obviously the problem just lends itself to the use of GUI.

A third usability feature is the inclusion of there being a list of vector field in a database that you can browse and plot. This is useful to the user because it gives them the ability to come back to their plot later as well as share it with others. If there were no method for saving your vector field, then the user would have to write it down in some other form which is inconvenient. And of course, this feature is needed for the notion of teachers sharing the plot with their students. The fact that the vector fields are shown as a list is a UX decision to make everything clear to the user. If they were not shown as a list but you had to input an exact name or code or something to get to the plot you want, then maybe it would be better off in the first place for the user to just remember the vector function of the plot they wanted.

Development

State machine implementation

As explained in the design section, I have decided that a state machine implementation will be the best way to go about this. The code used to implement this is boilerplate code, which can't really be meaningfully tested as it's being written but must be tested holistically due to how the classes are composed and aggregated from each other as shown in the UML class diagram. So, I will go ahead until the creation of the main menu state class and test things from there.

State.hpp

```cpp
#pragma once

class State {
public:
	virtual void init() = 0;
	virtual void handle_input() = 0;
	virtual void update(float dt) = 0;
	virtual void draw(float dt) = 0;

	virtual void pause() {}
	virtual void resume() {}
};
```

StateMachine.hpp

```cpp
#pragma once

#include <memory>
#include <stack>
#include "State.hpp"

typedef std::unique_ptr<State> state_ref;

class StateMachine {
public:
	StateMachine() {}
	~StateMachine() {}
	void push_state(state_ref new_state, bool is_replacing = true);
	void pop_state();
	void process();
	state_ref& get_active_state();

private:
	std::stack<state_ref> _states;
	state_ref _new_state;
	bool _is_removing;
	bool _is_adding;
	bool _is_replacing;
};
```

StateMachine.cpp

```cpp
#include "StateMachine.hpp"

void StateMachine::push_state(state_ref new_state, bool is_replacing) {
	this->_is_adding = true;
	this->_is_replacing = is_replacing;
	this->_new_state = std::move(new_state);
}

void StateMachine::pop_state() {
	this->_is_removing = true;
}

void StateMachine::process() {
	if (this->_is_removing && !this->_states.empty()) {
		this->_states.pop();

		if(!this->_states.empty()) {
			this->_states.top()->resume();
		}

		this->_is_removing = false;
	}

	if (this->_is_adding) {
		if (!this->_states.empty()) {
			if (this->_is_replacing) {
				this->_states.pop();
			} else {
				this->_states.top()->pause();
			}
		}

		this->_states.push(std::move(this->_new_state));
		this->_states.top()->init();
		this->_is_adding = false;
	}
}

state_ref& StateMachine::get_active_state() {
	return this->_states.top();
}
```

AssetManager.hpp

```
#pragma once

#include <map>
#include <SFML/Graphics.hpp>

class AssetManager {
public:
	AssetManager();
	~AssetManager();

	void load_texture(std::string name, std::string filename);
	sf::Texture& get_texture(std::string name);

	void load_font(std::string name, std::string filename);
	sf::Font& get_font(std::string name);

private:
	std::map<std::string, sf::Texture> _textures;
	std::map<std::string, sf::Font> _fonts;
};
```

AssetManager.cpp

```
#include "AssetManager.hpp"

void AssetManager::load_texture(std::string name, std::string filename) {
	sf::Texture texture;

	if (texture.loadFromFile(filename)) {
		this->_textures[name] = texture;
	}
}

sf::Texture& AssetManager::get_texture(std::string name) {
	return this->_textures.at(name);
}

void AssetManager::load_font(std::string name, std::string filename) {
	sf::Font font;

	if (font.loadFromFile(filename)) {
		this->_fonts[name] = font;
	}
}

sf::Font& AssetManager::get_font(std::string name) {
	return this->_fonts.at(name);
}
```

InputManager.hpp

```
#pragma once

#include <SFML/Graphics.hpp>

class InputManager {
public:
	InputManager() {};
	~InputManager() {};

	bool is_sprite_clicked(sf::Sprite object, sf::Mouse::Button button,
sf::RenderWindow& window);
	sf::Vector2i get_mouse_position(sf::RenderWindow& window);
};
```

InputManager.cpp

```
#include "InputManager.hpp"

bool InputManager::is_sprite_clicked(sf::Sprite object, sf::Mouse::Button button,
sf::RenderWindow& window) {
	if (sf::Mouse::isButtonPressed(button)) {
		sf::IntRect rect(object.getPosition().x, object.getPosition().y,
object.getGlobalBounds().width, object.getGlobalBounds().height);

		return rect.contains(sf::Mouse::getPosition(window));
	}

	return false;
}

sf::Vector2i InputManager::get_mouse_position(sf::RenderWindow& window) {
	return sf::Mouse::getPosition(window);
}
```

Program.hpp

```
#pragma once

#include <memory>
#include <string>
#include <SFML/Graphics.hpp>
#include "AssetManager.hpp"
#include "StateMachine.hpp"
#include "InputManager.hpp"

struct ProgramData {
	StateMachine state_machine;
	AssetManager asset_manager;
	InputManager input_manager;

	sf::RenderWindow window;
};

typedef std::shared_ptr<ProgramData> ProgramDataRef;

class Program {
public:
	Program(int width, int height, std::string title);
private:
	const float dt = 1.0f / 60.0f; // frame rate = 60 fps
	sf::Clock _clock;
	ProgramDataRef _data = std::make_shared<ProgramData>();

	void run();
};
```

Program.cpp
```
#include "Program.hpp"

Program::Program(int width, int height, std::string title) {
	_data->window.create(sf::VideoMode(width, height), title, sf::Style::Close |
sf::Style::Titlebar);
	_data->state_machine.push_state(state_ref(new Menu(this->_data)));

	this->run();
}
void Program::run() {
	float new_time;
	float frame_time;
	float interpolation;

	float current_time = this->_clock.getElapsedTime().asSeconds();
	float acc = 0.0f;

	while (this->_data->window.isOpen()) {
		this->_data->state_machine.process();
		new_time = this->_clock.getElapsedTime().asSeconds();

		frame_time = new_time - current_time;
		current_time = new_time;
		acc += frame_time;

		while (acc >= dt) {
			this->_data->state_machine.get_active_state()->handle_input();
			this->_data->state_machine.get_active_state()->update(dt);

			acc -= dt;
		}

		interpolation = acc / dt;
		this->_data->state_machine.get_active_state()->draw(interpolation);
	}
}
```

Menu.hpp
```
#pragma once

#include <SFML/Graphics.hpp>
#include "State.hpp"
#include "Program.hpp"

class Menu : public State {
public:
	Menu(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	sf::Clock _clock;
	sf::Sprite _background;
};
```

Menu.cpp

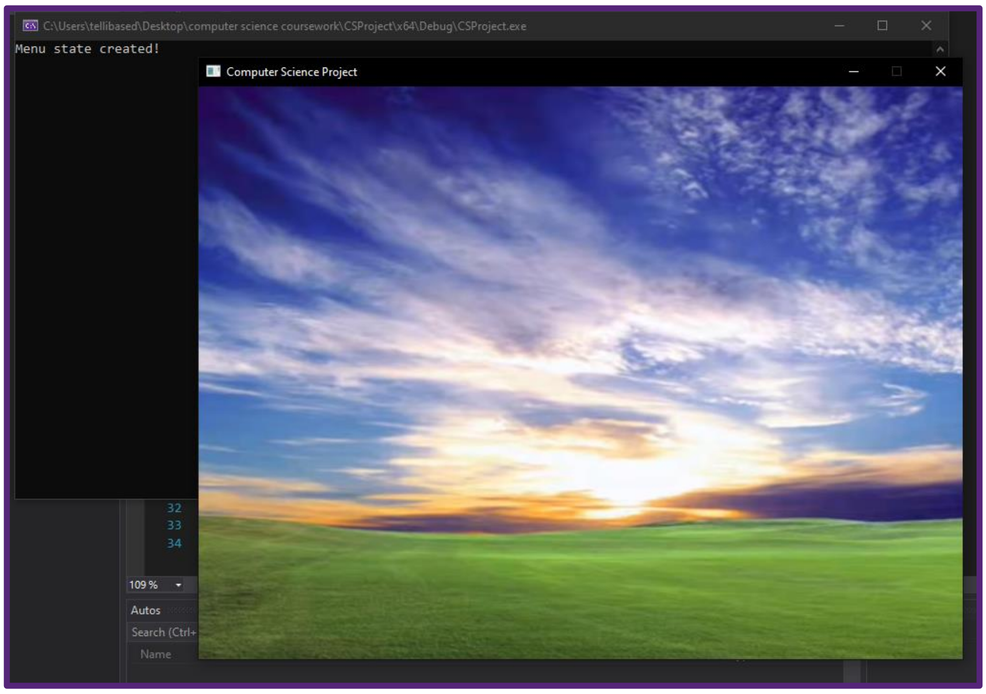
```
#include <sstream>
#include <iostream>
#include "Menu.hpp"

Menu::Menu(ProgramDataRef data) {
	this->_data = data;
}

void Menu::init() {
	std::cout << "Menu state created!" << std::endl;

	this->_data->asset_manager.load_texture("bg_menu", "bg_menu.PNG");
	_background.setTexture(this->_data->asset_manager.get_texture("bg_menu"));
}

void Menu::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) {
		if (event.type == sf::Event::Closed) {
			this->_data->window.close();
		}
	}
}

void Menu::update(float dt) {

}

void Menu::draw(float dt) {
	this->_data->window.clear();
	this->_data->window.draw(this->_background);
	this->_data->window.display();
}
```

Now that I have a functioning menu state, I can test the program.

main.cpp
```cpp
#include <SFML/Graphics.hpp>
#include "Program.hpp"

int main()
{
    Program(800, 600, "Computer Science Project");

    return 0;
}
```

The instantiation of the Program object creates an instance of the Menu class which displays bg_menu.png (a placeholder image)

Compiling and running the program, I see the following on my screen:


C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Computer Science Project
109 %
Autos
Search (Ctrl+
Name

```
Menu state created!
```

For now, the program works as expected. It displays the placeholder image bg_menu.png. What I need to do at this point is create buttons (list, plot, exit) which the user can click to go to the next state. I will also add a textbox class, which will take in keyboard input (e.g. the vector function.) and I will get rid of the menu placeholder image.

Button.hpp

```
#pragma once

#include <iostream>
#include <SFML/Graphics.hpp>

class Button {
public:
	Button(std::string button_text, sf::Vector2f size, int char_size, sf::Color 
bg_color, sf::Color text_color);
	void set_font(sf::Font& font);
	void set_back_color(sf::Color color);
	void set_text_color(sf::Color color);
	void set_position(sf::Vector2f pos);
	void draw(sf::RenderWindow& window);
	bool is_mouse_over(sf::RenderWindow& window);
private:
	sf::RectangleShape button;
	sf::Text text;
};
```

Button.cpp

```cpp
#include "Button.hpp"

Button::Button(std::string button_text, sf::Vector2f size, int char_size, sf::Color
bg_color, sf::Color text_color) {
	this->text.setString(button_text);
	this->text.setFillColor(text_color);
	this->text.setCharacterSize(char_size);

	this->button.setSize(size);
	this->button.setFillColor(bg_color);
}

void Button::set_font(sf::Font& font) {
	this->text.setFont(font);
}

void Button::set_back_color(sf::Color color) {
	this->button.setFillColor(color);
}

void Button::set_text_color(sf::Color color) {
	this->text.setFillColor(color);
}

void Button::set_position(sf::Vector2f pos) {
	this->button.setPosition(pos);

	float x_pos = pos.x + (button.getLocalBounds().width -
text.getLocalBounds().width) / 2;
	float y_pos = pos.y + (button.getLocalBounds().height -
text.getLocalBounds().height) / 2;

	this->text.setPosition({ x_pos, y_pos - text.getCharacterSize()/4});
}

void Button::draw(sf::RenderWindow& window) {
	window.draw(this->button);
	window.draw(this->text);
}

bool Button::is_mouse_over(sf::RenderWindow& window) {
	float mouse_x = sf::Mouse::getPosition(window).x;
	float mouse_y = sf::Mouse::getPosition(window).y;

	float button_x = this->button.getPosition().x;
	float button_y = this->button.getPosition().y;

	float button_pos_width = this->button.getPosition().x + this-
>button.getLocalBounds().width;
	float button_pos_height= this->button.getPosition().y + this-
>button.getLocalBounds().height;

	return (mouse_x < button_pos_width && mouse_x > button_x && mouse_y <
button_pos_height && mouse_y > button_y);
}
```

I have also changed Menu.hpp and Menu.cpp to accommodate the buttons:

Menu.hpp
```
#pragma once

#include <SFML/Graphics.hpp>
#include "State.hpp"
#include "Program.hpp"
#include "Button.hpp"

class Menu : public State {
public:
	Menu(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	sf::Clock _clock;
	sf::Sprite _background;
	Button plot_button = Button("Plot", { 80, 40 }, 20, sf::Color(72, 126, 242),
sf::Color::White);
	Button list_button = Button("List", { 80, 40 }, 20, sf::Color(72, 126, 242),
sf::Color::White);
	Button exit_button = Button("Exit", { 80, 40 }, 20, sf::Color(72, 126, 242),
sf::Color::White);
};
```

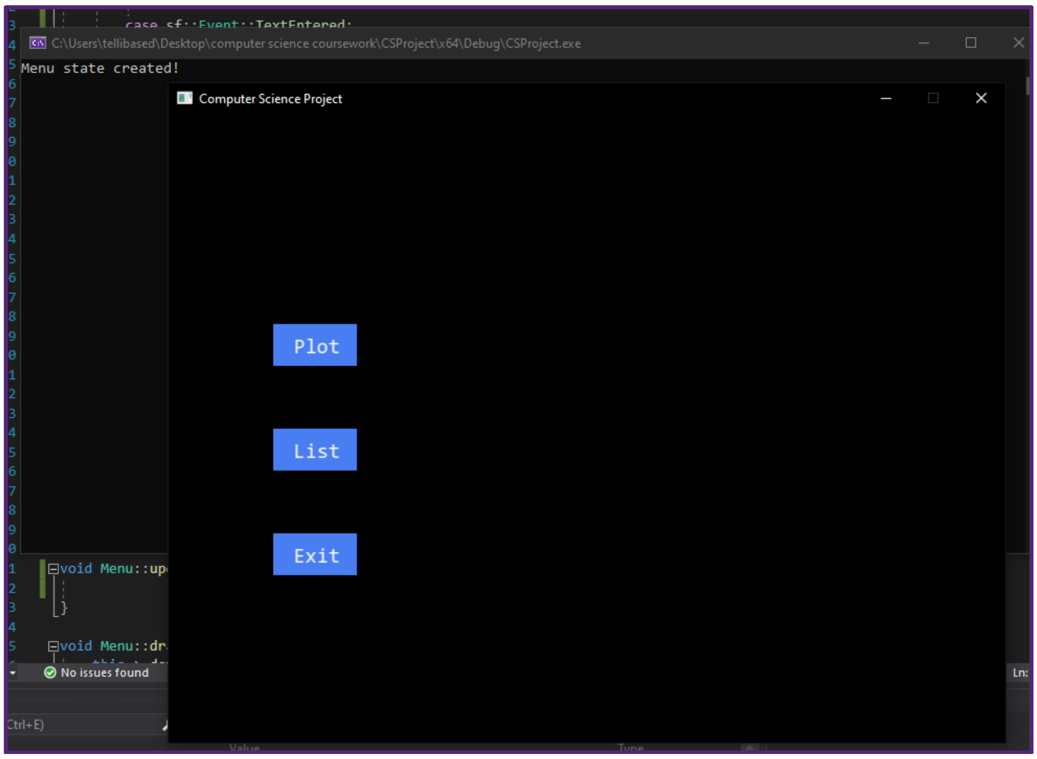
Menu.cpp

```
#include <sstream>
#include <iostream>
#include "Menu.hpp"

Menu::Menu(ProgramDataRef data) {
	this->_data = data;

	this->_data->asset_manager.load_font("consolas", "consola.ttf");
	this->plot_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->list_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->exit_button.set_font(this->_data->asset_manager.get_font("consolas"));

	this->plot_button.set_position({ 100, 200 });
	this->list_button.set_position({ 100, 300 });
	this->exit_button.set_position({ 100, 400 });
}

void Menu::init() {
	std::cout << "Menu state created!" << std::endl;

	//this->_data->asset_manager.load_texture("bg_menu", "bg_menu.PNG");
	//_background.setTexture(this->_data->asset_manager.get_texture("bg_menu"));
}

void Menu::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();

		case sf::Event::MouseMoved:
			for (Button* b : {&plot_button, &list_button, &exit_button}) {
				// iterate over references to buttons
				if (b->is_mouse_over(this->_data->window)) {
					b->set_back_color(sf::Color(42, 96, 212));
					// change color if hovered over
				} else {
					b->set_back_color(sf::Color(72, 126, 242));
					// else change back
				}
			}
```

(continued)

```
            case sf::Event::MouseButtonReleased:
                if (event.mouseButton.button == sf::Mouse::Left) {
                // left button released
                    if (plot_button.is_mouse_over(this->_data->window)) {
                        std::cout << "Plot button clicked" << std::endl;
                    }else if(list_button.is_mouse_over(this->_data->window)) {
                        std::cout << "List button clicked" << std::endl;
                    }else if(exit_button.is_mouse_over(this->_data->window)) {
                        std::cout << "Exit button clicked" << std::endl;
                        this->_data->window.close();
                    }
                }

            }
        }
}

void Menu::update(float dt) {

}

void Menu::draw(float dt) {
        this->_data->window.clear();
        this->_data->window.draw(this->_background);
        this->plot_button.draw(this->_data->window);
        this->list_button.draw(this->_data->window);
        this->exit_button.draw(this->_data->window);
        this->_data->window.display();
}
```

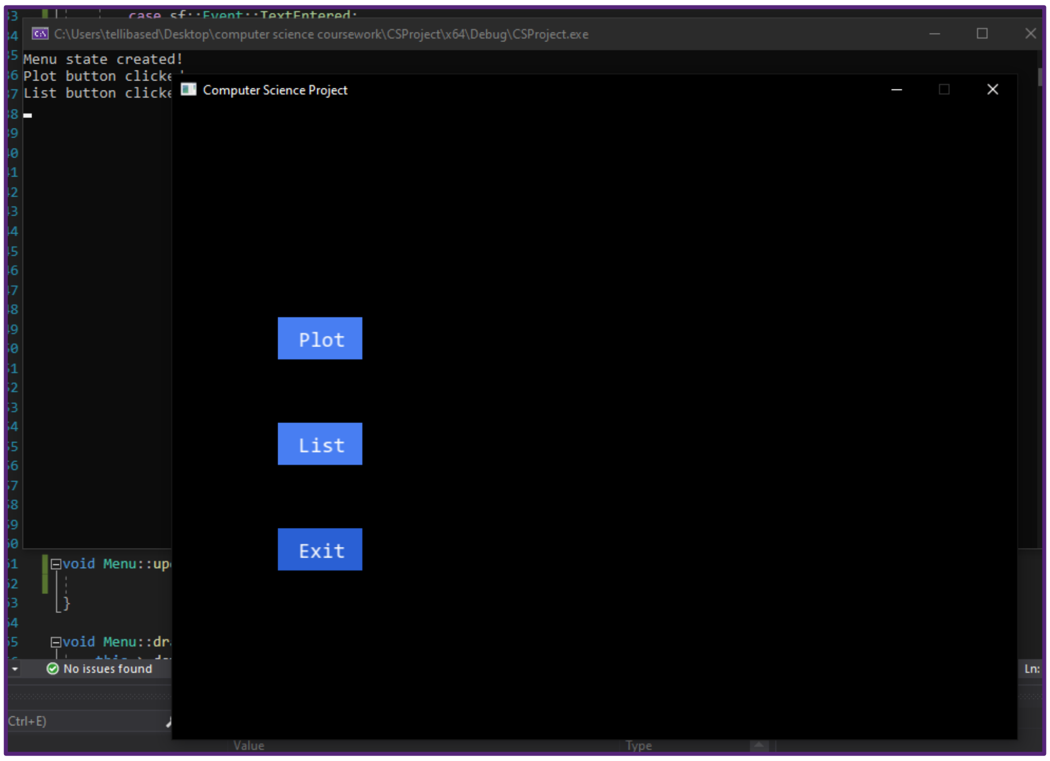
Running the program now, I now get this on my screen:



```
Menu state created!
```

C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Computer Science Project
Plot
List
Exit
No issues found

When hovering over a button, its colour changes (I have also clicked the plot and list buttons to make sure that works) (my mouse cursor is on the exit button, you can't see it for some reason though)



```
case sf::Event::TextEntered:
```

C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe

```
Menu state created!
Plot button clicke
List button clicke
```

```
void Menu::up
}
void Menu::dr
```

Clicking the exit button closes the program as expected.

Now that the menu has basic functionality, I will create a basic Plot state, which will take in keyboard input for the vector function through two textboxes. This input will then be parsed using the shunting-yard algorithm, storing the vector function as a tree of numeric values (which includes 'x' and 'y') and operators.

In summary, I'm about to create:
-Textbox class
-Plot class
-Shunting-yard class

Textbox.hpp

```cpp
#pragma once

#include <iostream>
#include <SFML/Graphics.hpp>
#include <sstream>

#define BACKSPACE 8 // ascii code for backspace
#define ENTER 13

class Textbox {
public:
	Textbox(int size, sf::Color color, bool selected);
	void set_font(sf::Font& font);
	void set_position(sf::Vector2f pos);
	void set_limit(bool limited);
	void set_limit(int limit);
	void set_selected(bool selected);
	void set_text(std::string str);
	std::string get_text();
	void draw(sf::RenderWindow& window);
	void on_type(sf::Event event);
private:
	sf::Text textbox;
	std::ostringstream text;
	bool is_selected = false;
	bool has_limit = false;
	int limit;

	void input_logic(int char_typed);
	void delete_last_char();
};
```

Textbox.cpp

```cpp
#include "Textbox.hpp"

Textbox::Textbox(int size, sf::Color color, bool selected) {
	this->textbox.setCharacterSize(size);
	this->textbox.setFillColor(color);
	this->is_selected = selected;

	if (selected) {
		textbox.setString("_");
	} else {
		textbox.setString("");
	}
}

void Textbox::input_logic(int char_typed) {
	if (char_typed == BACKSPACE) {
		if (text.str().length() > 0) {
			this->delete_last_char();
		}
	} else if (char_typed != ENTER) {
		text << static_cast<char>(char_typed);
	}

	textbox.setString(text.str() + "_");
}

void Textbox::delete_last_char() {
	std::string temp = text.str();
	std::string new_text = "";

	for (int i = 0; i < temp.length() - 1; i++) {
		//add all characters except for last one
		new_text += temp[i];
	}

	text.str(""); // clear current text
	text << new_text; // append to text
}

void Textbox::set_font(sf::Font& font) {
	this->textbox.setFont(font);
}

void Textbox::set_position(sf::Vector2f pos) {
	this->textbox.setPosition(pos);
}

void Textbox::set_limit(bool limited) {
	this->has_limit = limited;
}

void Textbox::set_limit(int limit) {
	this->has_limit = true;
	this->limit = limit;
}

void Textbox::set_selected(bool selected) {
	this->is_selected = selected;

	if (!selected) {
        // if its not selected we dont want the _ at the end
        std::string temp = text.str();
        std::string new_text = "";

        for (int i = 0; i < temp.length() - 1; i++) {
            //add all characters except for last one
            new_text += temp[i];
        }
        textbox.setString(new_text);
    }
}

void Textbox::set_text(std::string str) {
    text.str(""); // clear texsdt stream to nothing
    text << str; // append full string to new stream
    textbox.setString(str+"_"); // set displayed text to the string
}

std::string Textbox::get_text() {
    return this->text.str();
}

void Textbox::draw(sf::RenderWindow& window) {
    window.draw(textbox);
}

void Textbox::on_type(sf::Event event) {
    if (this->is_selected) {
        int char_typed = event.text.unicode;
        if (char_typed < 128) {
            // allow ascii only
            if (this->has_limit) {
                if (text.str().length() <= limit) {
                    this->input_logic(char_typed);
                }
                else if(text.str().length() > limit && char_typed ==
            BACKSPACE) {
                //allow them to delete chracters when over the limit
                    this->delete_last_char();
                }
            } else {
                this->input_logic(char_typed);
            }
        }
    }
}
```

Plot.hpp

```
#pragma once

#include <SFML/Graphics.hpp>
#include "State.hpp"
#include "Program.hpp"
#include "Button.hpp"
#include "Textbox.hpp"

class Plot : public State {
public:
	Plot(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	sf::Clock _clock;
	std::string text_one;
	std::string text_two;
	int current_component = 1;
	Textbox function_entry = Textbox(25, sf::Color(72, 126, 242), true);
	sf::Text prompt;
	sf::Text state_title;
};
```

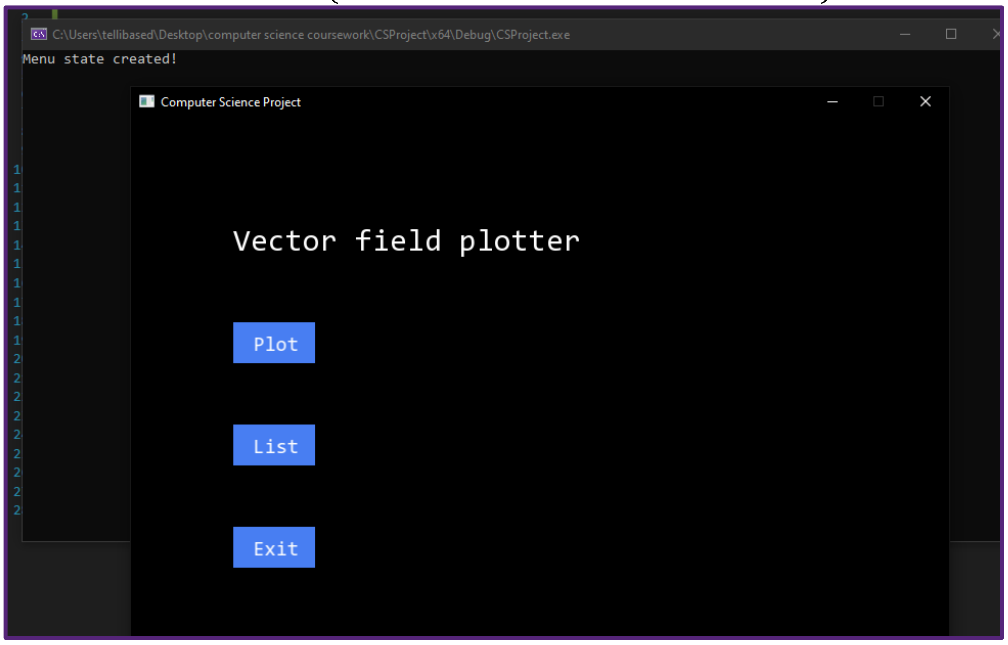
Plot.cpp

```
#include "Plot.hpp"

Plot::Plot(ProgramDataRef data) {
	this->_data = data;

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}

void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) {
		// get events that happen (user inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
			} else {
				std::cout << function_entry.get_text() << std::endl;
				if (current_component == 1) {
				prompt.setString("Enter second component of vector
function:");
					this->text_one = function_entry.get_text();
					this->function_entry.set_text("");
					current_component++;
				} else if (current_component == 2) {
					std::cout << "Begin display" << std::endl;
				}
			}
		}
	}
}

void Plot::update(float dt) {

}
```

```
void Plot::draw(float dt) {
	this->_data->window.clear();
	this->function_entry.draw(this->_data->window);
	this->_data->window.draw(prompt);
	this->_data->window.draw(state_title);
	this->_data->window.display();
}
```

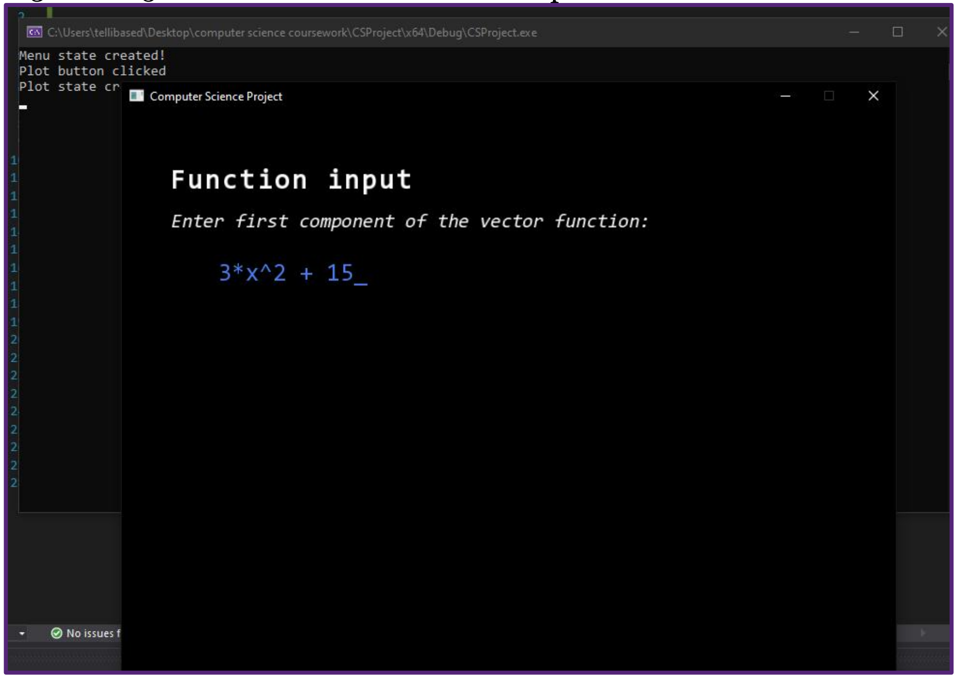
This is a placeholder version of the plot class which only takes in user input for the vector field. Here is how the program is going so far.

1). The main menu is functional. (I also added a title to the main menu.)



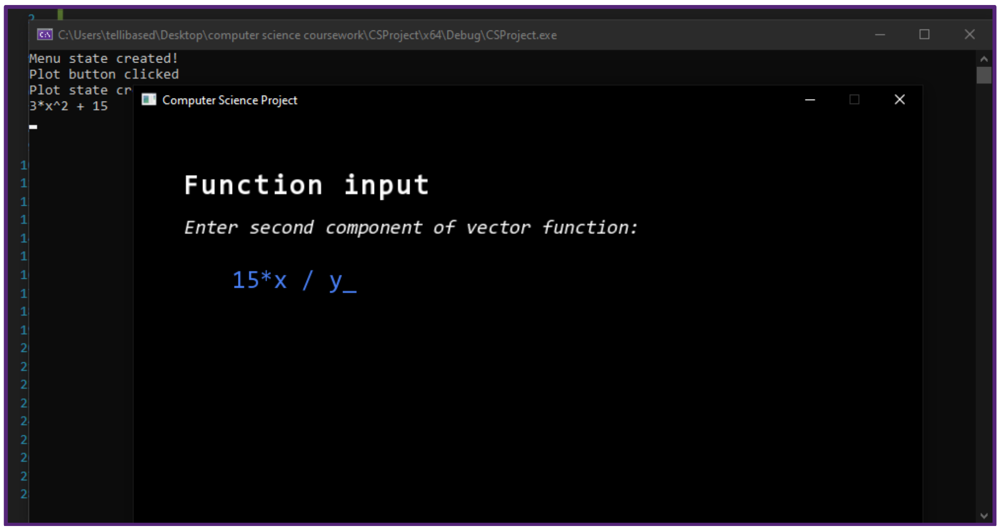
```
C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Menu state created!
```

2). I then clicked the Plot button which took me to the plot state. I entered the first component 3x^2 + 15 of the vector field I wanted to plot.



```
C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Menu state created!
Plot button clicked
Plot state cr
```

3). Then I pressed enter to confirm the first component. It then prompts me for the second component. I typed in 15*x / y which represents 15x/y.
The vector function I would be trying to plot here is f(x,y) = (3x^2 + 15, 15x/y).



```
Menu state created!
Plot button clicked
Plot state cr
3*x^2 + 15
```

Function input

Enter second component of vector function:

```
15*x / y_
```

C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Computer Science Project

In order to parse the function that was input by the user, I need to implement the shunting-yard algorithm into the program. This is probably the lengthiest module of the project. The implementation, which comes from a tutorial I found online, will take up 10 pages; it includes infix-to-postfix parsing (postfix is called RPN in the code, i.e., reverse polish notation) as well as postfix evaluation.

Relating this to the analysis section, we currently have three of the nine success criterions (the main menu with buttons, user can input vector function, input is taken from keyboard)

ShuntingYard.hpp

```
#ifndef SHUNTING_YARD
#define SHUNTING_YARD

#include <vector>
#include <string>
#include <map>
#include <stack>
#include <cmath>

namespace ShuntingYard {
	/*
		Typedefs
	*/

	// RPN list
	typedef std::vector<std::string> RPN;

	// callback to unary function (1 argument)
	typedef double(*UnaryFuncEval)(double x);

	// callback to binary function (2 arguments)
	typedef double(*BinaryFuncEval)(double x, double y);

	// types
	enum class TokenTypes {
		OPERATOR,
		CONSTANT,
		FUNCTION,
		LPAREN,
		RPAREN,
		ELSE
	};

	/*
		Utility function callbacks
	*/

	// determine if vector contains values
	template<typename T>
	bool contains(std::vector<T> v, T x);

	// obtain key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m);

	// obtain combined key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m1,
std::map<std::string, T> m2);

	// determine if character is number
	bool isNumber(char c, bool acceptDecimal = true, bool acceptNegative = true);

	// determine if entire string is number
	bool isNumber(const char* str);

	// determine if string only contains numerical characters
	bool containsNumbers(const char* str);

	// get numerical value of string
	double getNumericalVal(const char* str, float x, float y);

	// determine if string matches a function
	bool isFunction(std::string str);

	// determine if function is left associative
	bool isLeftAssociative(std::string str);

	// get function precedence
	short getPrecedence(std::string str);

	// find element from list in the equation starting at index i
	std::string findElement(int i, const char* eqn, std::vector<std::string> list);

	/*
		Function class definition
	*/
	class Func {
	public:
		// default constructor
		Func();

			// constructor for unary functions
		Func(UnaryFuncEval eval, TokenTypes type = TokenTypes::FUNCTION, short
prec = 0, bool left = true);

		// constructor for binary functions
		Func(BinaryFuncEval eval, TokenTypes type = TokenTypes::FUNCTION, short
prec = 0, bool left = true);

		double eval(double x, double y = 0);

		UnaryFuncEval u_eval;		// unary function evaluation callback
		BinaryFuncEval b_eval;	// binary function evaluation callback

		TokenTypes type;		// type of function (ie function or operator)
		short prec;				// precedence
		bool left;				// is left associative
		bool unary;				// is a unary function

	private:
		Func(TokenTypes type, short prec, bool left, bool unary);
	};

	/*
		Reference
	*/

	// unary functions


	/*
		Node class definitions
	*/

	// base node class
	class Node {
	public:
		Node(std::string name, bool isFunc);

		double eval(double x = 0, double y = 0);
```

```
        std::string name;
        bool isFunc;

        Node* right;
        Node* left;
};

// function node class
class FuncNode : public Node {
public:
        FuncNode(std::string name);

        // set type of function and then assign callback
        void setUnary(bool isUnary);

        // evaluate
        double eval(double x, double y = 0);

        bool isUnary;
        Func func;
};

// number node class
class NumNode : public Node {
public:
        NumNode(std::string name);

        // return numerical value
        double eval(double x = 0, double y = 0);
};

/*
        Main functions
*/

// parse infix notation into reverse polish notation (Shunting Yard)
RPN reversePolishNotation(const char* eqn);

// parse RPN to tree
Node* parse(RPN rpn);

// evaluate tree
double eval(Node* tree, float x, float y);

/*
        Utility function definitions
*/
```

```
	// determine if vector contains values
	template<typename T>
	bool contains(std::vector<T> v, T x);

	// obtain key list
	template<typename T>
	inline std::vector<std::string> keys(std::map<std::string, T> m);

	// obtain combined key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m1, 
std::map<std::string, T> m2);

	// determine if character is number
	bool isNumber(char c, bool acceptDecimal, bool acceptNegative);

	// determine if entire string is number
	bool isNumber(const char* str);

	// determine if string only contains numerical characters
	bool containsNumbers(const char* str);

	// get numerical value of string
	double getNumericalVal(const char* str, float x, float y);

	// determine if string matches a function
	bool isFunction(std::string str);

	// determine if function is left associative
	bool isLeftAssociative(std::string str);

	// get function precedence
	short getPrecedence(std::string str);

	// find element from list in the equation starting at index i
	std::string findElement(int i, const char* eqn, std::vector<std::string> list);
}

#endif
```

ShuntingYard.cpp

```
#include "ShuntingYard.hpp"
#include <iostream>

namespace ShuntingYard {
	std::map<std::string, Func> unary_functions = {
		{ "sin", Func(std::sin) }
	};
	// binary functions
	std::map<std::string, Func> binary_functions = {
		{ "+", Func([](double x, double y) -> double { return x + y; },
TokenTypes::OPERATOR, 2) },
		{ "-", Func([](double x, double y) -> double { return x - y; },
TokenTypes::OPERATOR, 2) },
		{ "*", Func([](double x, double y) -> double { return x * y; },
TokenTypes::OPERATOR, 3) },
		{ "/", Func([](double x, double y) -> double { return x / y; },
TokenTypes::OPERATOR, 3) },
		{ "^", Func(std::pow, TokenTypes::OPERATOR, 4, false) }
	};

	// function names
	std::vector<std::string> functionNames = keys<Func>(unary_functions,
binary_functions);

	// constants
	std::map<std::string, double> constants = {
		{ "pi", std::atan(1) * 4 },
		{ "e", std::exp(1) }
	};

	// constant names
	std::vector<std::string> constantNames = keys<double>(constants);

	// variables
	std::map<std::string, double> variables;

	// operators
	std::vector<char> operators = { '+', '-', '/', '*', '^' };
	// left brackets
	std::vector<char> leftBrackets = { '(', '{', '[' };
	// right brackets
	std::vector<char> rightBrackets = { ')', '}', ']' };

	Func::Func()
		: type(TokenTypes::OPERATOR), prec(0), left(true), unary(true),
u_eval(nullptr), b_eval(nullptr) {}

	Func::Func(UnaryFuncEval eval, TokenTypes type, short prec, bool left)
		: Func(type, prec, left, true) {
		u_eval = eval;
	}

	Func::Func(BinaryFuncEval eval, TokenTypes type, short prec, bool left)
		: Func(type, prec, left, false) {
		b_eval = eval;
	}
```

```
double Func::eval(double x, double y) {
		return this->unary ? u_eval(x) : b_eval(x, y);
	}

	Func::Func(TokenTypes type, short prec, bool left, bool unary)
		: type(type), prec(prec), left(left), unary(unary), u_eval(nullptr),
b_eval(b_eval) {}

	Node::Node(std::string name, bool isFunc)
		: name(name), isFunc(isFunc) {}

	FuncNode::FuncNode(std::string name)
		: Node(name, true) {}

	void FuncNode::setUnary(bool isUnary) {
		this->isUnary = isUnary;

		this->func = isUnary ? unary_functions[name] : binary_functions[name];
	}

	double FuncNode::eval(double x, double y) {
		return this->func.eval(x, y);
	}

	NumNode::NumNode(std::string name)
		: Node(name, false) {}

	double NumNode::eval(double x, double y) {
		return getNumericalVal(name.c_str(), x, y);
	}

	RPN reversePolishNotation(const char* eqn) {
		std::vector<std::string> queue;
		std::stack<std::string> stack;

		std::string obj = "";
		TokenTypes type = TokenTypes::ELSE;
		TokenTypes prevType = TokenTypes::ELSE; // negative sign detection

		bool acceptDecimal = true;
		bool acceptNegative = true;

		// token reading and detection
		for (int i = 0, eqLen = (int)strlen(eqn); i < eqLen; i++) {
			char t = eqn[i];

			// skip spaces and commas
			if (t == ' ' || t == ',') {
				//prevType = TokenTypes::ELSE;
				continue;
			}
```

```
            // classify token
            if (isNumber(t)) {
                type = TokenTypes::CONSTANT;
                if (t == '.') {
                    acceptDecimal = false;
                }
                else if (t == '-') {
                    acceptNegative = false;
                }

                int startI = i;
                if (i < eqLen - 1) {
                    while (isNumber(eqn[i + 1], acceptDecimal,
acceptNegative)) {
                        i++;
                        if (i >= eqLen - 1) {
                            break;
                        }
                    }
                }
                obj = std::string(eqn).substr(startI, i - startI + 1);

                // subtraction sign detection
                if (obj == "-") {
                    type = TokenTypes::OPERATOR;
                }
            }
            else {
                obj = findElement(i, eqn, functionNames);
                if (obj != "") {
                    // found valid object
                    type = contains<char>(operators, obj[0]) ?
TokenTypes::OPERATOR : TokenTypes::FUNCTION;
                }
                else {
                    obj = findElement(i, eqn, constantNames);
                    if (obj != "") {
                        // found valid object
                        type = TokenTypes::CONSTANT;
                    }
                    else {
                        obj = findElement(i, eqn, {"x", "y"});
                        if (obj != "") {
                            type = TokenTypes::CONSTANT;
                        }
                        else if (contains<char>(leftBrackets, t)) {
                            type = TokenTypes::LPAREN;
                            obj = "(";
                        }
                        else if (contains<char>(rightBrackets, t)) {
                            type = TokenTypes::RPAREN;
                            obj = ")";
                        }
                        else {
                            type = TokenTypes::ELSE;
                        }
                    }
                }
                i += obj.size() - 1;
            }
```

```
                    // classify token
                    if (isNumber(t)) {
                        type = TokenTypes::CONSTANT;
                        if (t == '.') {
                            acceptDecimal = false;
                        }
                        else if (t == '-') {
                            acceptNegative = false;
                        }

                        int startI = i;
                        if (i < eqLen - 1) {
                            while (isNumber(eqn[i + 1], acceptDecimal,
        acceptNegative)) {
                                i++;
                                if (i >= eqLen - 1) {
                                    break;
                                }
                            }
                        }
                        obj = std::string(eqn).substr(startI, i - startI + 1);

                        // subtraction sign detection
                        if (obj == "-") {
                            type = TokenTypes::OPERATOR;
                        }
                    }
                    else {
                        obj = findElement(i, eqn, functionNames);
                        if (obj != "") {
                            // found valid object
                            type = contains<char>(operators, obj[0]) ?
        TokenTypes::OPERATOR : TokenTypes::FUNCTION;
                        }
                        else {
                            obj = findElement(i, eqn, constantNames);
                            if (obj != "") {
                                // found valid object
                                type = TokenTypes::CONSTANT;
                            }
                            else {
                                obj = findElement(i, eqn, {"x", "y"});
                                if (obj != "") {
                                    type = TokenTypes::CONSTANT;
                                }
                                else if (contains<char>(leftBrackets, t)) {
                                    type = TokenTypes::LPAREN;
                                    obj = "(";
                                }
                                else if (contains<char>(rightBrackets, t)) {
                                    type = TokenTypes::RPAREN;
                                    obj = ")";
                                }
                                else {
                                    type = TokenTypes::ELSE;
                                }
                            }
                        }
                        i += obj.size() - 1;
                    }
```

```
// parse RPN to tree
	Node* parse(RPN rpn) {
		std::stack<Node*> stack;

		for (std::string item : rpn) {
			if (isNumber(item.c_str())) {
				// push number node
				stack.push(new NumNode(item));
			}
			else {
				// function
				FuncNode* f = new FuncNode(item);
				if (contains<std::string>(keys(binary_functions), item)) {
					f->setUnary(false);
					// set children of node

					// right child is second argument
					f->right = stack.top();
					stack.pop();

					// left child is first argument
					f->left = stack.top();
					stack.pop();
				}
				else if (contains<std::string>(keys(unary_functions),
item)) {
					f->setUnary(true);
					// set child of node
					f->left = stack.top();
					stack.pop();
				}
				stack.push(f);
			}
		}

		if (stack.size() == 0) {
			return nullptr;
		}

		return stack.top();
	}
```

```
// evaluate tree
	double eval(Node* tree, float x, float y) {
		if (tree->isFunc) {
			FuncNode* ftree = (FuncNode*)tree;
			if (ftree->isUnary) {
				// evaluate child recursively and then evaluate with
return value
				return ftree->eval(eval(tree->left, x, y));
			}
			else {
				// evaluate each child recursively and then evaluate with
return value
				return ftree->eval(eval(tree->left, x, y), eval(tree-
>right, x, y));
			}
		}
		else {
			// number node
			return ((NumNode*)tree)->eval(x, y);
		}
	}

	/*
		Utility function definitions
	*/

	// determine if vector contains values
	template<typename T>
	bool contains(std::vector<T> v, T x) {
		return std::find(v.begin(), v.end(), x) != v.end();
	}
```

```
// obtain key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m) {
		std::vector<std::string> ret;

		// push each key from each pair
		for (auto const& element : m) {
			ret.push_back(element.first);
		}

		return ret;
	}

	// obtain combined key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m1,
std::map<std::string, T> m2) {
		// get keys from each map
		std::vector<std::string> keySet1 = keys<T>(m1);
		std::vector<std::string> keySet2 = keys<T>(m2);

		// insert the second list into first
		keySet1.insert(keySet1.end(), keySet2.begin(), keySet2.end());

		// return result
		return keySet1;
	}

	// determine if character is number
	bool isNumber(char c, bool acceptDecimal, bool acceptNegative) {
		// digits
		if (c >= '0' && c <= '9') {
			return true;
		}
		// decimal point
		else if (acceptDecimal && c == '.') {
			return true;
		}
		// negative sign
		else if (acceptNegative && c == '-') {
			return true;
		}

		return false;
	}
```

```
// determine if entire string is number
	bool isNumber(const char* str) {
		// it's a constant, variable, or a numerical string
		return contains<std::string>(constantNames, str) ||
			contains<std::string>({"x", "y"}, str) ||
			containsNumbers(str);
	}

	// determine if string only contains numerical characters
	bool containsNumbers(const char* str) {
		// cannot be a single decimal point or negative sign
		if (std::strcmp(str, ".") == 0 || std::strcmp(str, "-") == 0) {
			return false;
		}

		std::string obj = std::string(str);

		// try to prove wrong
		bool acceptDecimal = true;
		if (isNumber(obj[0], true, true)) {
			// check first character for negative sign
			if (obj[0] == '.') {
				// cannot be any more decimal points
				acceptDecimal = false;
			}
		}
		else {
			return false;
		}

		for (unsigned int i = 1, len = obj.size(); i < len; i++) {
			// do not accept anymore negative signs
			if (!isNumber(obj[i], acceptDecimal, false)) {
				return false;
			}

			if (obj[i] == '.') {
				// cannot be any more decimal points
				acceptDecimal = false;
			}
		}

		return true;
	}

	// get numerical value of string
	double getNumericalVal(const char* str, float x = 0, float y = 0) {
		if (contains<std::string>(constantNames, str)) {
			// is a constant
			return constants[str];
		}
		else if (std::string(str) == "x") {
			return x;
		}
		else if (std::string(str) == "y") { return y; }
		else {
			// is a number
			return std::atof(str);
		}
	}
```

```
// determine if string matches a function
	bool isFunction(std::string str) {
		return contains<std::string>(functionNames, str);
	}

	// determine if function is left associative
	bool isLeftAssociative(std::string str) {
		return binary_functions[str].left;
	}

	// get function precedence
	short getPrecedence(std::string str) {
		if (contains<std::string>(keys(binary_functions), str)) {
			return binary_functions[str].prec;
		}

		// only care about operators, which are binary functions, so otherwise 
we can return 0
		return 0;
	}

	// find element from list in the equation starting at index i
	std::string findElement(int i, const char* eqn, std::vector<std::string> list) 
{
		for (std::string item : list) {
			int n = (int)item.size();
			if (std::string(eqn).substr(i, n) == item) {
				return item;
			}
		}

		return "";
	}
}
```

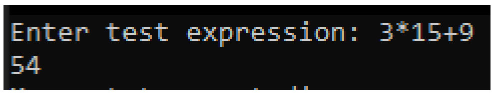
Now that it is implemented, I will test it using some basic expressions (which won't include x or y.) These were decided in the design section.
I will try the following:

1) *3 * 15 + 9*, which should output 54,
2) *sin(pi/2)*, which should output 1,
3) *e^(1/2)*, which should output 1.648 (to 3 decimal places),
4) *1 - 2*, which should output -1

main.cpp

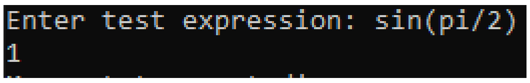
```cpp
#include <SFML/Graphics.hpp>
#include "Program.hpp"
#include "ShuntingYard.hpp"
#include <iostream>

int main()
{
    std::cout << "Enter test expression: ";

    std::string eqn;
    std::getline(std::cin, eqn);

    ShuntingYard::RPN rpn = ShuntingYard::reversePolishNotation(eqn.c_str());
    ShuntingYard::Node* tree = ShuntingYard::parse(rpn);
    std::cout << ShuntingYard::eval(tree) << std::endl;

    Program program = Program(800, 600, "Computer Science Project");

    return 0;
}
```

First test:



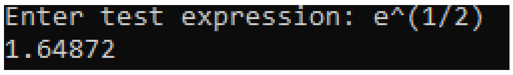
```
Enter test expression: 3*15+9
54
```

Second:



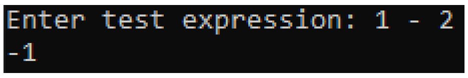
```
Enter test expression: sin(pi/2)
1
```

Third:



```
Enter test expression: e^(1/2)
1.64872
```

Fourth:



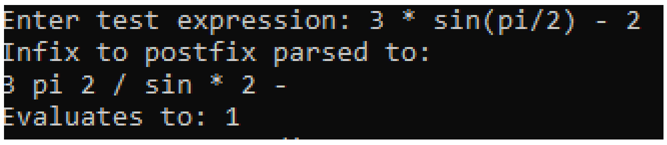
```
Enter test expression: 1 - 2
-1
```

I'm going to implement post-order traversal so that I can see what the input is actually being parsed into. This will let me know for certain that it's correct.

Main.hpp

```
#include <SFML/Graphics.hpp>
#include "Program.hpp"
#include "ShuntingYard.hpp"
#include <iostream>

void postorder(ShuntingYard::Node* root) {
    if (root == nullptr) return;

       postorder(root->left);
       postorder(root->right);

    std::cout << root->name << " ";
}

int main()
{
    std::cout << "Enter test expression: ";

    std::string eqn;
    std::getline(std::cin, eqn);

    ShuntingYard::RPN rpn = ShuntingYard::reversePolishNotation(eqn.c_str());
    ShuntingYard::Node* tree = ShuntingYard::parse(rpn);

    std::cout << "Infix to postfix parsed to:" << std::endl;
    postorder(tree);
    std::cout << std::endl;

    std::cout << "Evaluates to: " << ShuntingYard::eval(tree) << std::endl;

    Program program = Program(800, 600, "Computer Science Project");

    return 0;
}
```

Now running it:



```
Enter test expression: 3 * sin(pi/2) - 2
Infix to postfix parsed to:
3 pi 2 / sin * 2 -
Evaluates to: 1
```

This is correct.

The next thing to do is implement the 4th-order Runge Kutta method. This is not too involved; it was outlined in the design section.

RK4.hpp

```
#pragma once

#include "ShuntingYard.hpp"
#include <map>
#include <SFML/Graphics.hpp>

class Integrator {
public:
	Integrator(int steps, float end_time, ShuntingYard::Node* x_component,
ShuntingYard::Node* y_component, sf::Vector2f initial);
	void integrate();
	sf::Vector2f eval_vector_function(sf::Vector2f pos);
	sf::Vector2f get_position(float time);
private:
	std::map<float, sf::Vector2f> solution; // the solution (position at given
time)
	ShuntingYard::Node* x_component;	// x component of vector function
	ShuntingYard::Node* y_component;	// y component of vector function
	int steps;				// number of integration steps to be
taken
	float end_time;				 // time when simulation ends
};
```

RK4.cpp

```
#include "RK4.hpp"

Integrator::Integrator(int steps, float end_time,
	ShuntingYard::Node* x_component,
	ShuntingYard::Node* y_component,
	sf::Vector2f initial) {
	this->steps = steps;
	this->end_time = end_time;
	this->x_component = x_component;
	this->y_component = y_component;
	this->solution[0.0f] = initial;
}

sf::Vector2f Integrator::eval_vector_function(sf::Vector2f pos) {
	ShuntingYard::variables["x"] = pos.x;
	ShuntingYard::variables["y"] = pos.y;

	return { ShuntingYard::eval(this->x_component), ShuntingYard::eval(this-
>y_component) };
}

void Integrator::integrate() {
	//perform rk4
	float h = end_time / steps;

	for (int step = 0; step < steps; step++) {
		sf::Vector2f point_1 = this->eval_vector_function(solution[step * h]);
		sf::Vector2f point_2 = this->eval_vector_function(solution[step * h] +
(h/2)*point_1);
		sf::Vector2f point_3 = this->eval_vector_function(solution[step * h] +
(h/2)*point_2);
		sf::Vector2f point_4 = this->eval_vector_function(solution[step * h] +
h*point_3);

		solution[h * (step + 1)] = solution[h * step] + (h / 6) * (point_1 + 2.f
* point_2 + 2.f * point_3 + point_4);
	}
}

sf::Vector2f Integrator::get_position(float time) {
	if (time > end_time) {
		std::cout << "attempt to get position out of integration range" <<
std::endl;
	}

	float h = end_time / steps;
	float corrected_time = h * floor(time / h); // if h=0.1 then the pos for t=1.73
wont exist, round to 1.7

	return solution[corrected_time];
}
```

Now I will test the RK4 integrator with the test function f(x,y) = (3 * x + 4 * y, x), with an initial position of (1, 1). I will test to get the position at time t=0.1. By solving analytically, the solution is (1.8440, 1.1396) to 4 decimal places. To test & debug, I will quickly add some code to the Plot state which will output the position at t=0.1 of a particle whose original position was (1,1).

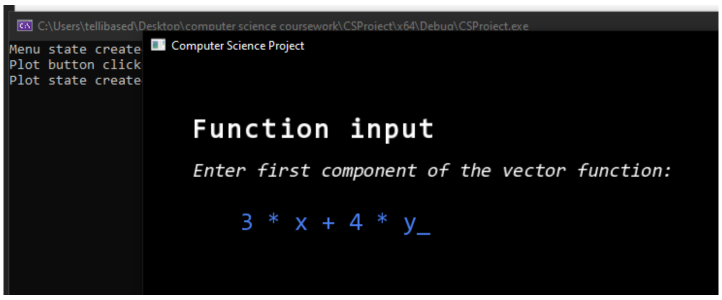
Plot.cpp

```
#include "Plot.hpp"
#include <iostream>

Plot::Plot(ProgramDataRef data) {
	this->_data = data;

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}
```

(continued)

```
void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
			} else {
				std::cout << function_entry.get_text() << std::endl;
				if (current_component == 1) {
					prompt.setString("Enter second component of vector
function:");

					ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
					this->x_component = ShuntingYard::parse(rpn);

					this->function_entry.set_text("");

					current_component++;
				} else if (current_component == 2) {
					ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
					this->y_component = ShuntingYard::parse(rpn);

					std::cout << "Begin display" << std::endl;

					Integrator integrator(100, 0.2f, this->x_component,
this->y_component, { 1.f, 1.f });
					integrator.integrate();

					sf::Vector2f test_solution =
integrator.get_position(0.1f);
					std::cout << "(" << test_solution.x << ", " <<
test_solution.y << ")" << std::endl;
				}
			}
		}
	}
}

void Plot::update(float dt) {

}

void Plot::draw(float dt) {
	this->_data->window.clear();
	this->function_entry.draw(this->_data->window);
	this->_data->window.draw(prompt);
	this->_data->window.draw(state_title);
	this->_data->window.display();
}
```

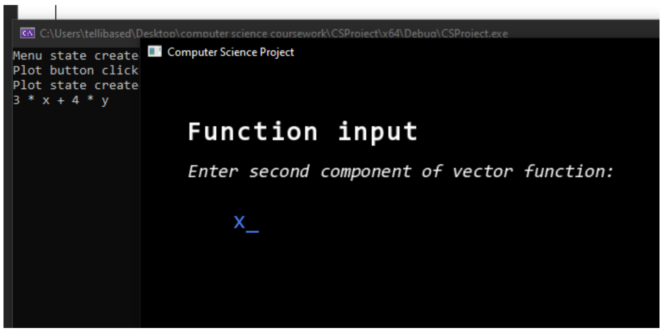
Input for first component:



```
C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Menu state create
Plot button click
Plot state create
```

```
Computer Science Project

Function input

Enter first component of the vector function:

    3 * x + 4 * y_
```

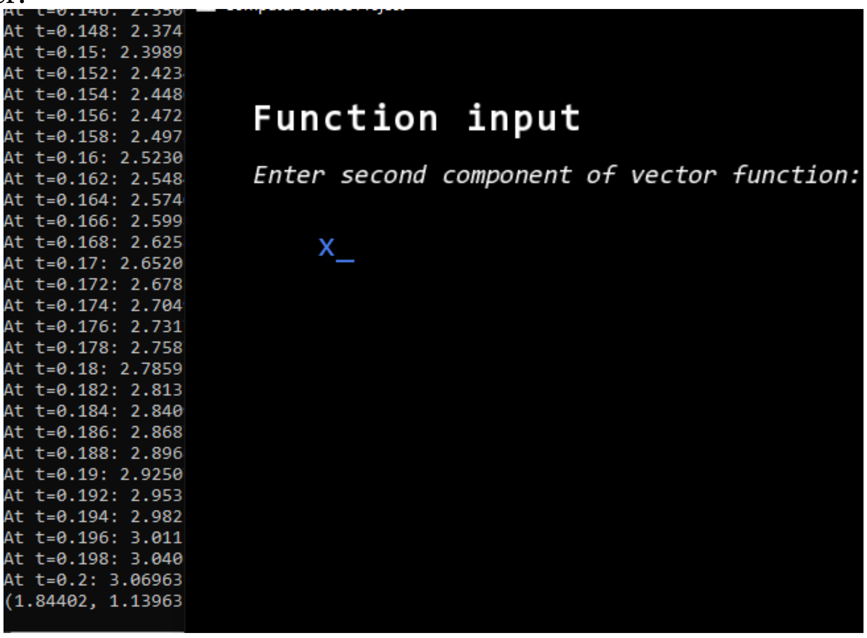
Second component:



```
C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Menu state create
Plot button click
Plot state create
3 * x + 4 * y
```

```
Computer Science Project

Function input

Enter second component of vector function:

    x_
```

Pressing enter:



```
At t=0.146: 2.350
At t=0.148: 2.374
At t=0.15: 2.3989
At t=0.152: 2.423
At t=0.154: 2.448
At t=0.156: 2.472
At t=0.158: 2.497
At t=0.16: 2.5230
At t=0.162: 2.548
At t=0.164: 2.574
At t=0.166: 2.599
At t=0.168: 2.625
At t=0.17: 2.6520
At t=0.172: 2.678
At t=0.174: 2.704
At t=0.176: 2.731
At t=0.178: 2.758
At t=0.18: 2.7859
At t=0.182: 2.813
At t=0.184: 2.840
At t=0.186: 2.868
At t=0.188: 2.896
At t=0.19: 2.9250
At t=0.192: 2.953
At t=0.194: 2.982
At t=0.196: 3.011
At t=0.198: 3.040
At t=0.2: 3.06963
(1.84402, 1.13963
```

```
Function input

Enter second component of vector function:

    x_
```

On the left you can see that the program is outputting the positions at times throughout the integration. At the end it outputs the time at t=0.1. The position matches up with the analytical solution so the test is successful, and we can conclude that the RK-4 algorithm has been implemented correctly.

Tying together the shunting-yard algorithm with an RK4 integrator was a lengthy but important part of the project; it is vital for the rest of the program.

Now I need to implement the moving particles. Each particle will have its own instance of the Integrator class because each particle has its own initial position on the grid, and therefore its own solution to the differential equation.

I will implement a Particle class which will contain its own integrator as well as some other properties. Then I will create some test code, which will display a static grid of particles.
Particle.hpp

```
#include "RK4.hpp"
#include <SFML/Graphics.hpp>

class Particle {
public:
	Particle(sf::Color color, float radius, int steps, int end_time, sf::Vector2f 
initial_position,
		ShuntingYard::Node* x_component, ShuntingYard::Node* y_component);
	void update_position(float time);
	void draw(sf::RenderWindow& window);
	sf::Vector2f get_position();
private:
	float radius;
	int steps;
	float end_time;
	sf::Vector2f initial_position;
	sf::CircleShape form;
	ShuntingYard::Node* x_component;
	ShuntingYard::Node* y_component;
	Integrator integrator = Integrator(steps, end_time, x_component, y_component, 
initial_position);
};
```

Particle.cpp

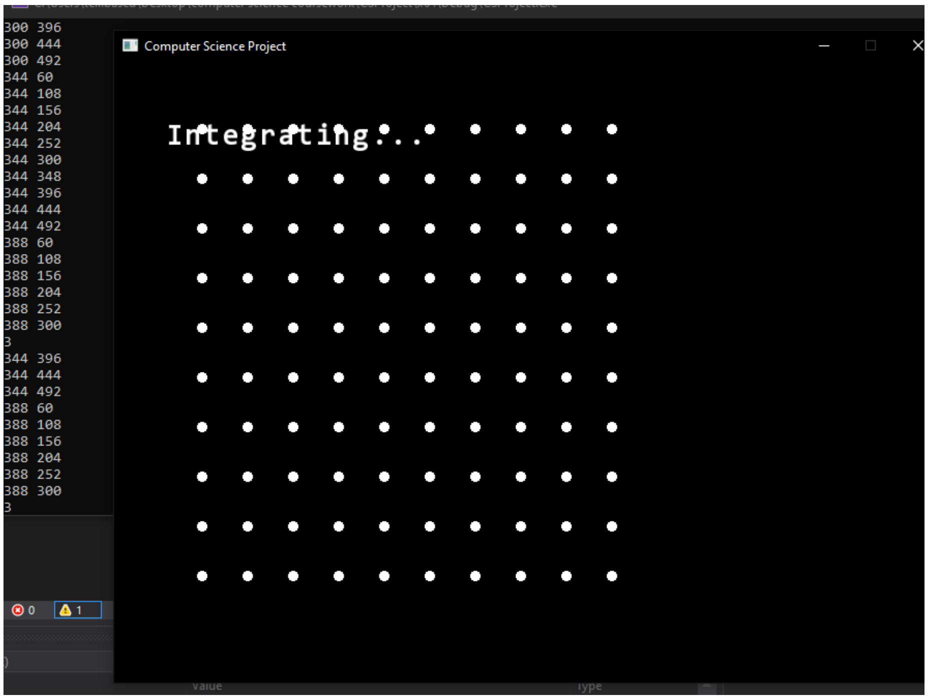
```
#include "Particle.hpp"

Particle::Particle(sf::Color color, float radius, int steps, int end_time, 
sf::Vector2f initial_position,
	ShuntingYard::Node* x_component, ShuntingYard::Node* y_component) {
	this->form.setFillColor(color);
	this->form.setRadius(radius);
	this->initial_position = initial_position;
	this->form.setPosition(initial_position);
	this->x_component = x_component;
	this->y_component = y_component;
	this->steps = steps;
	this->end_time = end_time;

	this->integrator = Integrator(steps, end_time, x_component, y_component, 
initial_position);
	this->integrator.integrate();
}

void Particle::update_position(float time) {
	this->form.setPosition(this->integrator.get_position(time));
}

void Particle::draw(sf::RenderWindow& window) {
	window.draw(this->form);
}

sf::Vector2f Particle::get_position() {
	return this->form.getPosition();
}
```

Plot.cpp

```
#include "Plot.hpp"
#include <iostream>

Plot::Plot(ProgramDataRef data) {
	this->_data = data;

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}

void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
			} else {
				std::cout << function_entry.get_text() << std::endl;
				if (current_component == 1) {
					prompt.setString("Enter second component of vector
function:");

					ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
					this->x_component = ShuntingYard::parse(rpn);

					this->function_entry.set_text("");

					current_component++;
				} else if (current_component == 2) {
					ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
					this->y_component = ShuntingYard::parse(rpn);

					/*std::cout << "Begin display" << std::endl;

					Integrator integrator(100, 0.2f, this->x_component,
this->y_component, { 1.f, 1.f });
					integrator.integrate();
```

```
                                        sf::Vector2f test_solution =
integrator.get_position(0.1f);
                                        std::cout << "(" << test_solution.x << ", " <<
test_solution.y << ")" << std::endl;*/

                                        function_entry.set_selected(false);
                                        current_component++;

                                        this->state_title.setString("Integrating...");
                                        this->draw(0); // show "Integrating" message before
initialising particles
                                        // if this weren't here it would run the below
block of code before re-drawing

                                        // create 2D grid of particles
                                        for (int i = 0; i < 10; i++) {
                                                for (int j = 0; j < 10; j++) {
                                                        Particle new_particle =
Particle(sf::Color::White, 5.f, 100, 1, {80.f + 44.f * i, 60.f + 48.f * j},
                                                this->x_component, this->y_component);

                                                this->particles.push_back(new_particle);
                                                }
                                        }
                                }
                        }
                }
        }
}

void Plot::update(float dt) {

}

void Plot::draw(float dt) {
        this->_data->window.clear();
        this->_data->window.draw(state_title);

        if (current_component <= 2) {
                // hide UI elements when finished inputting
                this->function_entry.draw(this->_data->window);
                this->_data->window.draw(prompt);
        }

        for (Particle p : this->particles) {
                p.draw(this->_data->window);
                std::cout << p.get_position().x << " " << p.get_position().y <<
std::endl;
        }

        this->_data->window.display();
}
```

Running it:



```
300 396
300 444
300 492
344 60
344 108
344 156
344 204
344 252
344 300
344 348
344 396
344 444
344 492
388 60
388 108
388 156
388 204
388 252
388 300
3
344 396
344 444
344 492
388 60
388 108
388 156
388 204
388 252
388 300
3
```

The console is displaying the positions of particles.

This test is a bit of a failure, because 1. the “integrating…” message is still showing while the particles are on screen; 2. the particles aren’t uniform throughout the screen (they’re all on the top left). I will fix this right now. The purpose of the “integrating…” message is to assure the user that the program hasn’t become frozen, but rather is running an algorithm.

I also want to change the colour of the UI for usability and aesthetic reasons.

I replaced the “2D grid of particles” block of code with this:

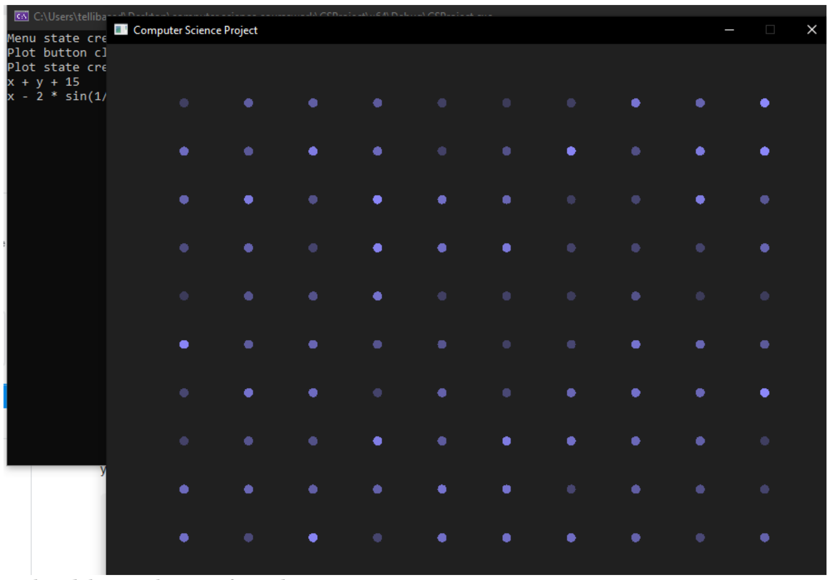
```
// create 2D grid of particles
                                   for (int i = 0; i < 10; i++) {
                                          for (int j = 0; j < 10; j++) {
                                   Particle new_particle = Particle(
                                   sf::Color(140, 136, 252, dis(gen)),
                                   5.f, 100, 1,
                                   {80.f + 71.1f * i, 60.f + 53.3f * j},
                                   this->x_component, this->y_component);

                                   this->particles.push_back(new_particle);
                                          }
                                   }

                                   this->state_title.setString("");
```

The new values for the initial position “`{80.f + 71.1f * i, 60.f + 53.3f * j}`” fixes the issue with the particles being non-uniform. The last line of the block `this->state_title.setString("");` fixes the issue of the “Integrating…” message still being displayed despite the particles being on screen.
This is unrelated to the two issues I just described, but I added this to the top of Plot.cpp, after the include statements:

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis(64, 255);
```

This is for random number generation. dis(64, 255) is used to generate random integers in the interval [64, 255]. In the block of code on the last page, I'm using this to generate random values for the alpha (transparency) of each particle; not only does it look nicer, but it will make it possible to see if particles are overlapping.

I also changed the first line of the draw function in the Plot class. It is now `this->_data->window.clear(sf::Color(32, 32, 32));` -- instead of clearing with a black screen its now a lighter colour.



```
Menu state cre
Plot button cl
Plot state cre
x + y + 15
x - 2 * sin(1/
```

The issues described have been fixed.
Now I need to make the particles move.
I first changed Plot.hpp and Plot.cpp:
Plot.hpp

```cpp
#pragma once

#include "Textbox.hpp"
#include "Program.hpp"
#include "RK4.hpp"
#include "Particle.hpp"

class Plot : public State {
public:
	Plot(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	float time_multiplier = 0.01;
	float time = 0;
	bool finished_integrating = false;
	ShuntingYard::Node* x_component;
	ShuntingYard::Node* y_component;
	int current_component = 1;
	Textbox function_entry = Textbox(25, sf::Color(72, 126, 242), true);
	sf::Text prompt;
	sf::Text state_title;
	std::vector<Particle> particles;
};
```

```
#include "Plot.hpp"
#include <iostream>
#include <random>

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis(64, 255);

Plot::Plot(ProgramDataRef data) {
	this->_data = data;

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}

void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
			} else {
				std::cout << function_entry.get_text() << std::endl;
				if (current_component == 1) {
					prompt.setString("Enter second component of vector
function:");

					ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
					this->x_component = ShuntingYard::parse(rpn);

					this->function_entry.set_text("");

					current_component++;
				} else if (current_component == 2) {
					ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
					this->y_component = ShuntingYard::parse(rpn);

					/*std::cout << "Begin display" << std::endl;
```

```
                                        Integrator integrator(100, 0.2f, this->x_component,
this->y_component, { 1.f, 1.f });
                                        integrator.integrate();

                                        sf::Vector2f test_solution =
integrator.get_position(0.1f);
                                        std::cout << "(" << test_solution.x << ", " <<
test_solution.y << ")" << std::endl;*/

                                        function_entry.set_selected(false);
                                        current_component++;

                                        this->state_title.setString("Integrating...");
                                        this->draw(0); // show "Integrating" message before
initialising particles
                                        // if this weren't here it would run the below
block of code before re-drawing

                                        // create 2D grid of particles
                                        for (int i = 0; i < 10; i++) {
                                                for (int j = 0; j < 10; j++) {
                                                        Particle new_particle =
Particle(sf::Color(140, 136, 252, dis(gen)), 5.f, 100, 10, {80.f + 71.1f * i, 60.f +
53.3f * j}, this->x_component, this->y_component);

                                                this->particles.push_back(new_particle);
                                                }
                                        }

                                        this->state_title.setString("");
                                        this->finished_integrating = true;
                                        this->time = 0;
                                }
                        }
                }
        }
}

void Plot::update(float dt) {
        this->time += dt * time_multiplier;

        if (finished_integrating) {
                for (Particle& p : particles) {
                        p.update_position(this->time);
                }
        }
}

void Plot::draw(float dt) {
        this->_data->window.clear(sf::Color(32, 32, 32));
        this->_data->window.draw(state_title);

        if (current_component <= 2) {
                // hide UI elements when finished inputting
                this->function_entry.draw(this->_data->window);
                this->_data->window.draw(prompt);
        }

        for (Particle& p : particles) {
                p.draw(this->_data->window);
        }
```

```
	this->_data->window.display();
}
```

Testing it, it works, but there is a problem. Due to the design of the `get_position` function of the integrator, the particles only move in a discrete number of frames rather than every frame. This is a problem when either h or the time multiplier is small (or both).

```
sf::Vector2f Integrator::get_position(float time) {
	if (time > end_time) {
		std::cout << "attempt to get position out of integration range" << 
std::endl;
	}

	float h = end_time / steps;
	float corrected_time = h * floor(time / h); // if h=0.1 then the pos for t=1.73 
wont exist, round to 1.7

	return solution[corrected_time];
}
```

I will make a video to illustrate the problem, but before that I will add text in the top left which says what the simulation time is.

(Added sf::Text simulation_time to private attributes in Plot.hpp)

Inside Plot constructor:
```
simulation_time.setFont(this->_data->asset_manager.get_font("consolas"));
simulation_time.setCharacterSize(15);
simulation_time.setPosition({ 20, 20 });
simulation_time.setFillColor(sf::Color(140, 136, 252));
```

Inside Update:
```
this->simulation_time.setString("time = " + std::to_string(this->time));
```

Draw:
```
if (finished_integrating) {
	this->_data->window.draw(simulation_time);
}
```

Here is the video:

https://youtu.be/hNNnRe_KH7Q

As you can see, the particles only move every 0.1 units of simulation time. This is a problem because the extremely small time multiplier (I've used 0.01) means the simulation only moves every 10 seconds. To fix this, I will implement linear interpolation, which means the particle will "slide" along a line from one point to another. This will make it so that the particle is moving every frame.

```
sf::Vector2f Integrator::get_position(float time) {
	/*if (time > end_time) {
		std::cout << "attempt to get position out of integration range" << 
std::endl;
		std::cout << time << end_time << std::endl;
	}*/

	float h = end_time / steps;
	float corrected_time = h * floor(time / h); // if h=0.1 then the pos for t=1.73 
wont exist, round to 1.7

	sf::Vector2f next_point = solution[h * (floor(time / h) + 1)];
	sf::Vector2f difference = next_point - solution[corrected_time];

	//std::cout << solution[corrected_time].x << ", " << solution[corrected_time].y 
<< std::endl;

	//as a ratio between 0 and 1
	float fraction_passed = ((time - h * floor(time / h)) / h);  // this is "how 
far" the particle has travelled along the path between the two points


	return solution[corrected_time] + fraction_passed * difference;
}
```

https://youtu.be/24KjYISrTN4

It now works well, but I noticed 2 more issues.

1) After integration, the simulation skips to a time t=(time multiplier) * (time taken to integrate). This is a real issue if the time taken to integrate is long.
2) For some functions the x and y positions will grow very large and will slow down the simulation.

Here is the fix I found for issue 1:

```
void Plot::update(float dt) {
	if (finished_integrating) {
		if (dt <= 1) {
			this->time += dt * time_multiplier;
			this->simulation_time.setString("time = " + std::to_string(this-
>time));
		}

		for (Particle& p : particles) {
			p.update_position(this->time);
		}
		this->draw(1.f / 60.f);
	}

}
```

A fix for issue 2 is to stop updating particle position if they are already offscreen.

```
void Plot::update(float dt) {
	if (finished_integrating) {
		if (dt <= 1) {
			this->time += dt * time_multiplier;
			this->simulation_time.setString("time = " + std::to_string(this-
>time));
		}

		for (Particle& p : particles) {
			sf::Vector2f posn = p.get_position();
			if (!(posn.x > 800.f || posn.y > 600.f || posn.x < -10.f ||
posn.y < -10.f)) {
				p.update_position(this->time);
			}
		}
		this->draw(1.f / 60.f);
	}

}
```

I added the menu button to the plot state.

Plot.hpp

```
#pragma once

#include "Textbox.hpp"
#include "Program.hpp"
#include "RK4.hpp"
#include "Particle.hpp"
#include "Button.hpp"

class Plot : public State {
public:
	Plot(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	float time_multiplier = 0.01;
	float time = 0;
	bool finished_integrating = false;
	ShuntingYard::Node* x_component;
	ShuntingYard::Node* y_component;
	int current_component = 1;
	Textbox function_entry = Textbox(25, sf::Color(140, 136, 252), true);
	Button menu_button = Button("Menu", { 80, 40 }, 20, sf::Color(140, 136, 252),
sf::Color::White);
	sf::Text prompt;
	bool flag = false;
	sf::Text state_title;
	sf::Text simulation_time;
	std::vector<Particle> particles;
};
```

Plot.cpp

```
#include "Plot.hpp"
#include <iostream>
#include <random>
#include "Menu.hpp"

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis(64, 255);

Plot::Plot(ProgramDataRef data) {
	this->_data = data;

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);

	simulation_time.setFont(this->_data->asset_manager.get_font("consolas"));
	simulation_time.setCharacterSize(15);
	simulation_time.setPosition({ 50, 20 });
	simulation_time.setFillColor(sf::Color(140, 136, 252));

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}

void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
			} else {
				std::cout << function_entry.get_text() << std::endl;
				if (current_component == 1) {
					prompt.setString("Enter second component of vector
function:");

					ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
					this->x_component = ShuntingYard::parse(rpn);
```

```
                                    this->function_entry.set_text("");

                                    current_component++;
                             } else if (current_component == 2) {
                                    ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(function_entry.get_text().c_str());
                                    this->y_component = ShuntingYard::parse(rpn);

                                    /*std::cout << "Begin display" << std::endl;

                                    Integrator integrator(100, 0.2f, this->x_component,
this->y_component, { 1.f, 1.f });
                                    integrator.integrate();

                                    sf::Vector2f test_solution =
integrator.get_position(0.1f);
                                    std::cout << "(" << test_solution.x << ", " <<
test_solution.y << ")" << std::endl;*/

                                    function_entry.set_selected(false);
                                    current_component++;

                                    this->state_title.setString("Integrating...");
                                    this->draw(0); // show "Integrating" message before
initialising particles
                                    // if this weren't here it would run the below
block of code before re-drawing

                                    // create 2D grid of particles
                                    for (int i = 0; i < 20; i++) {
                                           for (int j = 0; j < 20; j++) {
                                                  Particle new_particle =
Particle(sf::Color(140, 136, 252, dis(gen)), 3.f, 100, 1, {80.f + 35.1f * i, 60.f +
26.2f * j},
                                                         this->x_component, this-
>y_component);

                                                  new_particle.update_position(0);
                                                  this-
>particles.push_back(new_particle);

                                           }
                                    }

                                    this->state_title.setString("");
                                    this->finished_integrating = true;
                                    this->time = 0;
                             }
                      }
              case sf::Event::MouseMoved: // iterate over references to buttons
              {
                      if (menu_button.is_mouse_over(this->_data->window)) {
                             menu_button.set_back_color(sf::Color(140, 136, 252)); //
change color if hovered over
                      }
                      else {
                             menu_button.set_back_color(sf::Color(110, 106, 222)); //
else change back
                      }
              }
```

```
		case sf::Event::MouseButtonReleased:
			if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
				if (menu_button.is_mouse_over(this->_data->window)) {
					std::cout << "Menu button clicked" << std::endl;
					this->_data->state_machine.push_state(state_ref(new
Menu(_data)));
				}
			}

		}

	}
}

void Plot::update(float dt) {
	if (finished_integrating) {
		if (dt <= 1) {
			this->time += dt * time_multiplier;
			this->simulation_time.setString("time = " + std::to_string(this-
>time));
		}

		for (Particle& p : particles) {
			sf::Vector2f posn = p.get_position();
			if (!(posn.x > 800.f || posn.y > 600.f || posn.x < -10.f ||
posn.y < -10.f)) {
				p.update_position(this->time);
			}
		}
		this->draw(1.f / 60.f);
	}

}

void Plot::draw(float dt) {
	this->_data->window.clear(sf::Color(32, 32, 32));
	this->_data->window.draw(state_title);
	this->menu_button.draw(this->_data->window);

	if (current_component <= 2) {
		// hide UI elements when finished inputting
		this->function_entry.draw(this->_data->window);
		this->_data->window.draw(prompt);
	}

	if (finished_integrating) {
		for (Particle& p : particles) {
			p.draw(this->_data->window);
		}

		this->_data->window.draw(simulation_time);
	}

	this->_data->window.display();
}
```

To begin the implementation of the list functionality, I will create a class that can read and write to a csv file which will contain data for a vector field.

This class represents a vector field as a whole.
Field.hpp

```
#include "ShuntingYard.hpp"
#pragma once

#include <SFML/Graphics.hpp>

class Field {
public:
	Field(std::string x_component, std::string y_component, int steps, float
end_time, sf::Color color);
	Field(std::string name);
	void write(std::string name); // writes to file
private:
	std::string x_component;
	std::string y_component;
	int steps;
	float end_time;
	sf::Color color;
};
```

Field.cpp

```
#include "Field.hpp"
#include <fstream>

Field::Field(std::string x_component, std::string y_component, int steps, float
end_time, sf::Color color) {
	this->x_component = x_component;
	this->y_component = y_component;
	this->steps = steps;
	this->end_time = end_time;
	this->color = color;
}

void Field::write(std::string name) {
	std::ofstream fields_file;
	fields_file.open("fields.csv");

	fields_file << name << ";" << x_component << ";" << y_component
			   << ";" << steps << ";" << end_time << ";" << color.r
			<< ";" << color.g << ";" << color.b << "\n";

	fields_file.close();
}
```

I also need to add a Save button to the plot state, so that this new class can be tested. This involves a new iteration of Plot.hpp and Plot.cpp.

Plot.hpp

```
#pragma once

#include "Textbox.hpp"
#include "Program.hpp"
#include "RK4.hpp"
#include "Particle.hpp"
#include "Button.hpp"
#include "Field.hpp"

class Plot : public State {
public:
	Plot(ProgramDataRef data);
	Plot(ProgramDataRef data, Field basis);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	float time_multiplier = 0.01;
	float time = 0;
	int steps = 0;
	float end_time = 0;
	bool finished_integrating = false;
	std::string x_text;
	std::string y_text;
	ShuntingYard::Node* x_component;
	ShuntingYard::Node* y_component;
	int current_component = 1;
	Textbox function_entry = Textbox(25, sf::Color(140, 136, 252), true);
	Button menu_button = Button("Menu", { 80, 40 }, 20, sf::Color(140, 136, 252), 
sf::Color::White);
	sf::Text prompt;
	sf::Text state_title;
	sf::Text simulation_time;
	bool typing_name = false;
	std::vector<Particle> particles;
	Textbox naming = Textbox(25, sf::Color(140, 136, 252), false);
	Button save_button = Button("Save", { 80, 40 }, 20, sf::Color(140, 136, 252), 
sf::Color::White);
};
```

Plot.cpp

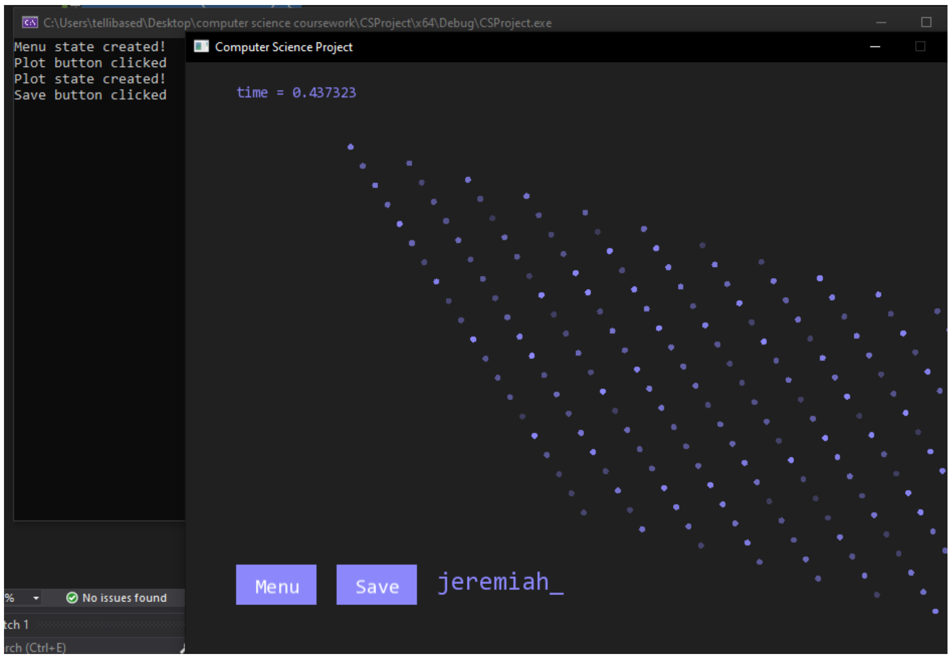
```
#include "Plot.hpp"
#include <iostream>
#include <random>
#include "Menu.hpp"

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis(64, 255);

Plot::Plot(ProgramDataRef data) {
	this->_data = data;

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	naming.set_font(this->_data->asset_manager.get_font("consolas"));
	naming.set_position({ 250, 500 });
	naming.set_limit(32);

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);

	simulation_time.setFont(this->_data->asset_manager.get_font("consolas"));
	simulation_time.setCharacterSize(15);
	simulation_time.setPosition({ 50, 20 });
	simulation_time.setFillColor(sf::Color(140, 136, 252));

	save_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->save_button.set_position({ 150, 500 });

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}

void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
				naming.on_type(event);
			} else {
```

```
                        if (current_component == 1) {
                              prompt.setString("Enter second component of vector
function:");

                              x_text = function_entry.get_text();

                              ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(x_text.c_str());
                              this->x_component = ShuntingYard::parse(rpn);

                              this->function_entry.set_text("");

                              current_component++;
                        } else if (current_component == 2) {
                              y_text = function_entry.get_text();

                              ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(y_text.c_str());
                              this->y_component = ShuntingYard::parse(rpn);

                              function_entry.set_selected(false);
                              current_component++;

                              this->state_title.setString("Integrating...");
                              this->draw(0); // show "Integrating" message before
initialising particles
                              // if this weren't here it would run the below
block of code before re-drawing

                              // create 2D grid of particles
                              for (int i = 0; i < 20; i++) {
                                    for (int j = 0; j < 20; j++) {
                                          Particle new_particle =
Particle(sf::Color(140, 136, 252, dis(gen)), 3.f, 100, 1, {80.f + 35.1f * i, 60.f +
26.2f * j},
                                                this->x_component, this-
>y_component);

                                          new_particle.update_position(0);
                                          this-
>particles.push_back(new_particle);

                                    }
                              }

                              this->state_title.setString("");
                              this->finished_integrating = true;
                              this->time = 0;
                        } else if (typing_name) {
                              // add to file
                              Field field = Field(x_text, y_text, 100, 1,
sf::Color(140, 136, 252));

                              field.write(naming.get_text());
                        }
                  }
            case sf::Event::MouseMoved: // iterate over references to buttons
            {
                  for (Button* b : { &menu_button, &save_button }) {
                        if (b->is_mouse_over(this->_data->window)) {
                              b->set_back_color(sf::Color(110, 106, 222));
                        } else {
```

```
                                        b->set_back_color(sf::Color(140, 136, 252));
                                }
                        }
                }


                case sf::Event::MouseButtonReleased:
                        if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
                                if (menu_button.is_mouse_over(this->_data->window)) {
                                        std::cout << "Menu button clicked" << std::endl;
                                        this->_data->state_machine.push_state(state_ref(new
Menu(_data)));
                                } else if (save_button.is_mouse_over(this->_data->window))
{
                                        std::cout << "Save button clicked" << std::endl;
                                        this->typing_name = true;
                                        this->naming.set_selected(true);
                                }
                        }

                }

        }
}

void Plot::update(float dt) {
        if (finished_integrating) {
                if (dt <= 1) {
                        this->time += dt * time_multiplier;
                        this->simulation_time.setString("time = " + std::to_string(this-
>time));
                }

                for (Particle& p : particles) {
                        sf::Vector2f posn = p.get_position();
                        if (!(posn.x > 800.f || posn.y > 600.f || posn.x < -10.f ||
posn.y < -10.f)) {
                                p.update_position(this->time);
                        }
                }
                this->draw(1.f / 60.f);
        }

}

void Plot::draw(float dt) {
        this->_data->window.clear(sf::Color(32, 32, 32));
        this->_data->window.draw(state_title);
        this->menu_button.draw(this->_data->window);

        if (current_component <= 2) {
                // hide UI elements when finished inputting
                this->function_entry.draw(this->_data->window);
                this->_data->window.draw(prompt);
        }

        if (finished_integrating) {
                for (Particle& p : particles) {
                        p.draw(this->_data->window);
                }
```

```
            this->_data->window.draw(simulation_time);
            this->save_button.draw(this->_data->window);
            if (this->typing_name) {
                this->naming.draw(this->_data->window);
            }
        }

        this->_data->window.display();
}
```

I will now test this by creating a vector field with x component x+y, y-component y-x and call it “jeremiah”.



```
C:\Users\tellibased\Desktop\computer science coursework\CSProject\x64\Debug\CSProject.exe
Menu state created!
Plot button clicked
Plot state created!
Save button clicked
```

Hitting enter, this is now fields.csv:

```
jeremiah;x + y;x - y;100;1;Œ;ˆ;ü
```

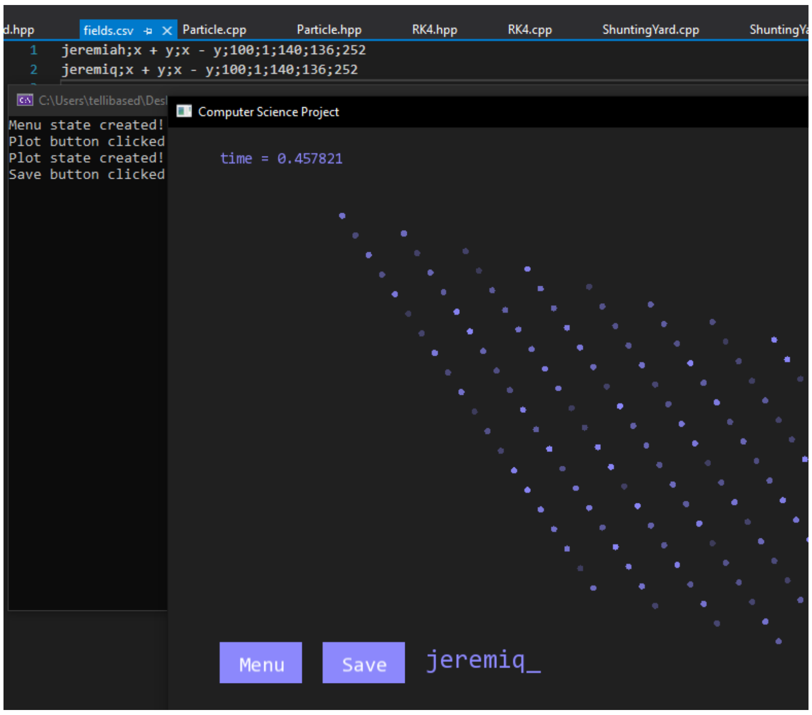
The test has failed. The expected contents should be:

```
jeremiah;x + y;x - y;100;1;140;136;252
```

I believe the nonsense Unicode characters appear because SFML internally uses its own implementation of integers which just so happens to not appear nicely when written to a file stream. All that has to be done to fix the error is to cast `color.r`, `color.g` and `color.b` to `int`.

I also noticed that the current contents will be overwritten every time instead of just appending – I will fix this as well.

```
void Field::write(std::string name) {
	std::ofstream fields_file;
	fields_file.open("fields.csv", std::ios_base::app);

	fields_file << name << ";" << x_component << ";" << y_component
			<< ";" << steps << ";" << end_time << ";" << (int)color.r
		    << ";" << (int)color.g << ";" << (int)color.b << "\n";

	fields_file.close();
}
```



```
jeremiah;x + y;x - y;100;1;140;136;252
jeremiq;x + y;x - y;100;1;140;136;252
```

```
Menu state created!
Plot button clicked
Plot state created!
Save button clicked
```

It now works as intended.
Now I need to add the ability to set up a simulation from a name. Before I do this, I need to give the user the ability to change some settings of the vector field; the end time and the number of steps per particle. This required yet another iteration of the Plot.hpp and Plot.cpp files.

Plot.hpp
```
#pragma once

#include "Textbox.hpp"
#include "Program.hpp"
#include "RK4.hpp"
#include "Particle.hpp"
#include "Button.hpp"
#include "Field.hpp"

class Plot : public State {
public:
	Plot(ProgramDataRef data);
	Plot(ProgramDataRef data, Field basis);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);

	void integrate();
private:
	ProgramDataRef _data;
	float time_multiplier = 0.01;
	float time = 0;
	float end_time = 0;
	int steps = 0;
	int current_input = 1;
	bool finished_integrating = false;
	bool typing_name = false;
	std::string x_text;
	std::string y_text;
	std::vector<Particle> particles;
	sf::Color particle_color;
	ShuntingYard::Node* x_component;
	ShuntingYard::Node* y_component;
	Textbox function_entry = Textbox(25, sf::Color(140, 136, 252), true);
	Textbox naming = Textbox(25, sf::Color(140, 136, 252), false);
	Button menu_button = Button("Menu", { 80, 40 }, 20, sf::Color(140, 136, 252), 
sf::Color::White);
	Button save_button = Button("Save", { 80, 40 }, 20, sf::Color(140, 136, 252), 
sf::Color::White);
	sf::Text prompt;
	sf::Text state_title;
	sf::Text simulation_time;
};
```

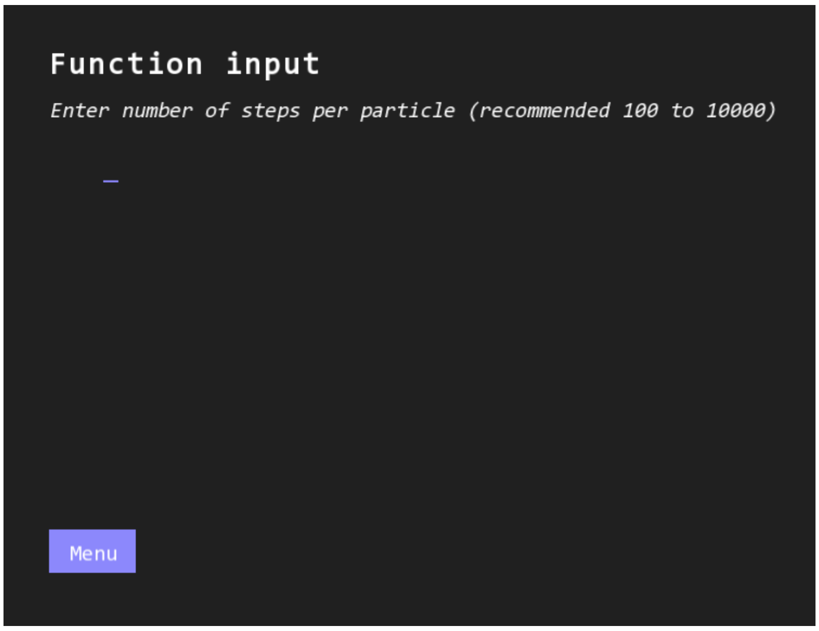
Plot.cpp

```
#include "Plot.hpp"
#include <iostream>
#include <random>
#include "Menu.hpp"

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis1(64, 255);
std::uniform_int_distribution<> dis2(0, 255);

Plot::Plot(ProgramDataRef data) {
	this->_data = data;
	this->particle_color = sf::Color(dis2(gen), dis2(gen), dis2(gen));

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	naming.set_font(this->_data->asset_manager.get_font("consolas"));
	naming.set_position({ 250, 500 });
	naming.set_limit(32);

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);

	simulation_time.setFont(this->_data->asset_manager.get_font("consolas"));
	simulation_time.setCharacterSize(15);
	simulation_time.setPosition({ 50, 20 });
	simulation_time.setFillColor(sf::Color(140, 136, 252));

	save_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->save_button.set_position({ 150, 500 });

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });
}

void Plot::integrate() {
	this->state_title.setString("Integrating...");
	this->draw(0); // show "Integrating" message before initialising particles
	// if this weren't here it would run the below block of code before re-drawing

	// create 2D grid of particles
	for (int i = 0; i < 20; i++) {
		for (int j = 0; j < 20; j++) {
			Particle new_particle = Particle(sf::Color(particle_color.r,
particle_color.g, particle_color.b, dis1(gen)),
                                                                              3.f,
steps, end_time, { 80.f + 35.1f * i, 60.f + 26.2f * j },
				this->x_component, this->y_component);

			new_particle.update_position(0);
			this->particles.push_back(new_particle);
```

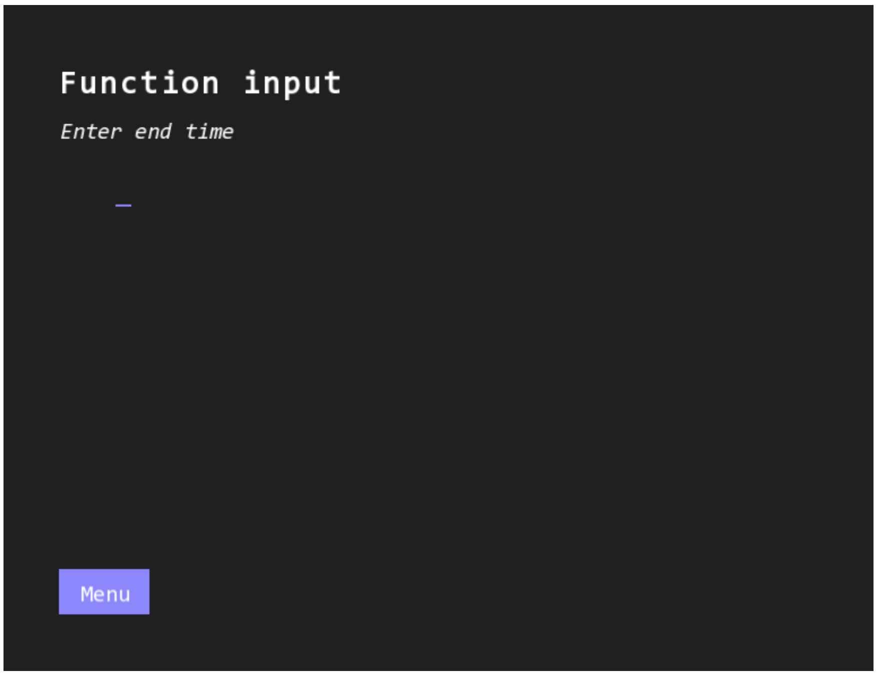
```
				}
			}

			this->state_title.setString("");
			this->finished_integrating = true;
			this->time = 0;
		}

		void Plot::init() {
			std::cout << "Plot state created!" << std::endl;
		}

		void Plot::handle_input() {
			sf::Event event;

			while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
				switch (event.type) {
				case sf::Event::Closed: // clicks x in top right
					this->_data->window.close();
				case sf::Event::TextEntered:
					if (event.text.unicode != ENTER) {
						function_entry.on_type(event);
						naming.on_type(event);
					} else {
						if (current_input == 1) {
							prompt.setString("Enter second component of vector
function:");

							x_text = function_entry.get_text();

							ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(x_text.c_str());
							this->x_component = ShuntingYard::parse(rpn);

							this->function_entry.set_text("");

							current_input++;
						} else if (current_input == 2) {
							y_text = function_entry.get_text();

							ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(y_text.c_str());
							this->y_component = ShuntingYard::parse(rpn);

							this->function_entry.set_text("");

							prompt.setString("Enter number of steps per
particle (recommended 100 to 10000)");

							current_input++;
						} else if (current_input == 3) {
							steps = stoi(function_entry.get_text());

							prompt.setString("Enter end time");

							this->function_entry.set_text("");

							current_input++;

						}
```

```
                        else if (current_input == 4) {
                                end_time = stof(function_entry.get_text());

                                function_entry.set_selected(false);
                                current_input++;

                                std::cout << end_time << std::endl;
                                std::cout << steps << std::endl;

                                integrate();
                        } else if (typing_name) {
                                // add to file
                                Field field = Field(x_text, y_text, steps,
end_time, sf::Color(particle_color.r, particle_color.g, particle_color.b));

                                field.write(naming.get_text());
                        }
                }
        case sf::Event::MouseMoved: // iterate over references to buttons
        {
                for (Button* b : { &menu_button, &save_button }) {
                        if (b->is_mouse_over(this->_data->window)) {
                                b->set_back_color(sf::Color(110, 106, 222));
                        } else {
                                b->set_back_color(sf::Color(140, 136, 252));
                        }
                }
        }


        case sf::Event::MouseButtonReleased:
                if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
                        if (menu_button.is_mouse_over(this->_data->window)) {
                                std::cout << "Menu button clicked" << std::endl;
                                this->_data->state_machine.push_state(state_ref(new
Menu(_data)));
                        } else if (save_button.is_mouse_over(this->_data->window))
{
                                std::cout << "Save button clicked" << std::endl;
                                this->typing_name = true;
                                this->naming.set_selected(true);
                        }
                }

        }

    }
}

void Plot::update(float dt) {
    if (finished_integrating) {
        if (dt <= 1) {
                this->time += dt * time_multiplier;
                this->simulation_time.setString("time = " + std::to_string(this-
>time));
        }

        for (Particle& p : particles) {
                sf::Vector2f posn = p.get_position();
                if (!(posn.x > 800.f || posn.y > 600.f || posn.x < -10.f ||
posn.y < -10.f)) {
```

```
                                p.update_position(this->time);
                        }
                }
                this->draw(1.f / 60.f);
        }

}

void Plot::draw(float dt) {
        this->_data->window.clear(sf::Color(32, 32, 32));
        this->_data->window.draw(state_title);
        this->menu_button.draw(this->_data->window);

        if (current_input <= 4) {
                // hide UI elements when finished inputting
                this->function_entry.draw(this->_data->window);
                this->_data->window.draw(prompt);
        }

        if (finished_integrating) {
                for (Particle& p : particles) {
                        p.draw(this->_data->window);
                }

                this->_data->window.draw(simulation_time);
                this->save_button.draw(this->_data->window);
                if (this->typing_name) {
                        this->naming.draw(this->_data->window);
                }
        }

        this->_data->window.display();
}
```

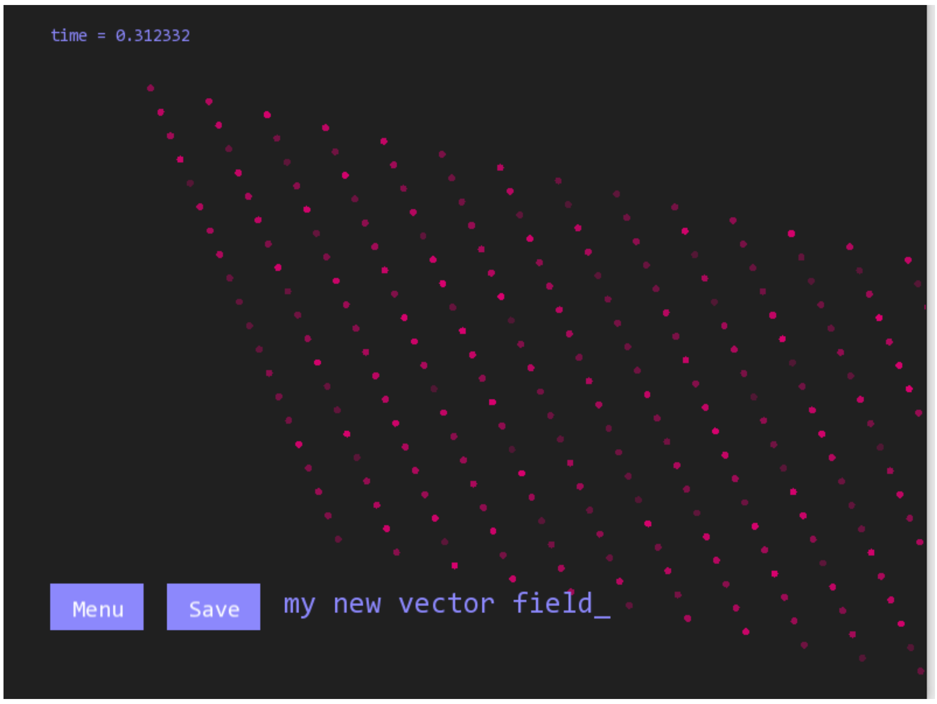

Function input

Enter number of steps per particle (recommended 100 to 10000)


Function input

Enter end time

_


time = 0.312332

my new vector field_

fields.csv
```
my new vector field;x + y;x - y;100;1;216;1;110
```

It looks like everything is going well. I will now proceed to add functionality to create a simulation from a Field object and then create the List state to conclude development.

Plot.cpp

```
#include "Plot.hpp"
#include <iostream>
#include <random>
#include "Menu.hpp"

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis1(64, 255);
std::uniform_int_distribution<> dis2(0, 255);

Plot::Plot(ProgramDataRef data) {
	this->_data = data;
	this->particle_color = sf::Color(dis2(gen), dis2(gen), dis2(gen));

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	naming.set_font(this->_data->asset_manager.get_font("consolas"));
	naming.set_position({ 250, 500 });
	naming.set_limit(32);

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);

	simulation_time.setFont(this->_data->asset_manager.get_font("consolas"));
	simulation_time.setCharacterSize(15);
	simulation_time.setPosition({ 50, 20 });
	simulation_time.setFillColor(sf::Color(140, 136, 252));

	save_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->save_button.set_position({ 150, 500 });

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });
}

Plot::Plot(ProgramDataRef data, Field field) {
	this->created_from_field = true;

	this->_data = data;
	this->particle_color = field.color;
	this->end_time = field.end_time;
	this->steps = field.steps;
	this->current_input = 5;

	std::cout << field.x_component << std::endl;
	std::cout << field.y_component << std::endl;

	this->x_component = 
ShuntingYard::parse(ShuntingYard::reversePolishNotation(field.x_component.c_str()));
```

```
	this->y_component = 
ShuntingYard::parse(ShuntingYard::reversePolishNotation(field.y_component.c_str()));

	integrate();
}

void Plot::integrate() {
	this->state_title.setString("Integrating...");
	this->draw(0); // show "Integrating" message before initialising particles
	// if this weren't here it would run the below block of code before re-drawing

	// create 2D grid of particles
	for (int i = 0; i < 20; i++) {
		for (int j = 0; j < 20; j++) {
			Particle new_particle = Particle(sf::Color(particle_color.r, 
particle_color.g, particle_color.b, dis1(gen)),
												3.f, 
steps, end_time, { 80.f + 35.1f * i, 60.f + 26.2f * j },
				this->x_component, this->y_component);

			new_particle.update_position(0);
			this->particles.push_back(new_particle);

		}
	}

	this->state_title.setString("");
	this->finished_integrating = true;
	this->time = 0;
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}

void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user 
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
				naming.on_type(event);
			} else {
				if (current_input == 1) {
					prompt.setString("Enter second component of vector 
function:");

					x_text = function_entry.get_text();

					ShuntingYard::RPN rpn = 
ShuntingYard::reversePolishNotation(x_text.c_str());
					this->x_component = ShuntingYard::parse(rpn);

					this->function_entry.set_text("");

					current_input++;
				} else if (current_input == 2) {
```

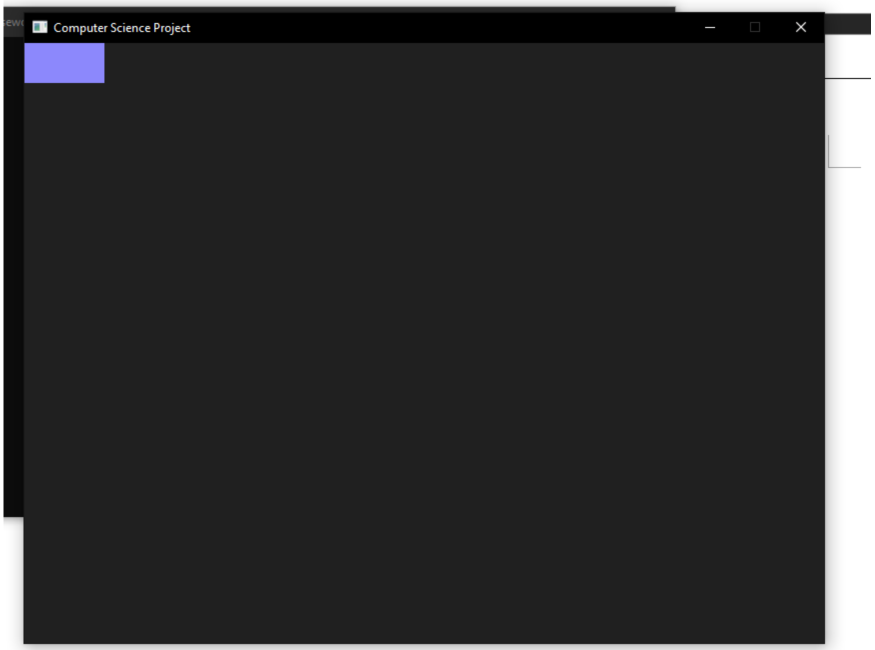
```
                                    y_text = function_entry.get_text();

                                    ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(y_text.c_str());
                                    this->y_component = ShuntingYard::parse(rpn);

                                    this->function_entry.set_text("");

                                    prompt.setString("Enter number of steps per
particle (recommended 100 to 10000)");

                                    current_input++;
                             } else if (current_input == 3) {
                                    steps = stoi(function_entry.get_text());

                                    prompt.setString("Enter end time");

                                    this->function_entry.set_text("");

                                    current_input++;

                             }
                             else if (current_input == 4) {
                                    end_time = stof(function_entry.get_text());

                                    function_entry.set_selected(false);
                                    current_input++;

                                    std::cout << end_time << std::endl;
                                    std::cout << steps << std::endl;

                                    integrate();
                             } else if (typing_name) {
                                    // add to file
                                    Field field = Field(x_text, y_text, steps,
end_time, sf::Color(particle_color.r, particle_color.g, particle_color.b));

                                    field.write(naming.get_text());
                             }
                      }
              case sf::Event::MouseMoved: // iterate over references to buttons
              {
                      for (Button* b : { &menu_button, &save_button }) {
                             if (b->is_mouse_over(this->_data->window)) {
                                    b->set_back_color(sf::Color(110, 106, 222));
                             } else {
                                    b->set_back_color(sf::Color(140, 136, 252));
                             }
                      }
              }


              case sf::Event::MouseButtonReleased:
                      if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
                             if (menu_button.is_mouse_over(this->_data->window)) {
                                    std::cout << "Menu button clicked" << std::endl;
                                    this->_data->state_machine.push_state(state_ref(new
Menu(_data)));
                             } else if (save_button.is_mouse_over(this->_data->window))
{
                                    std::cout << "Save button clicked" << std::endl;
```

```
                                        this->typing_name = true;
                                        this->naming.set_selected(true);
                                }
                        }

                }

        }
}

void Plot::update(float dt) {
        if (finished_integrating) {
                if (dt <= 1) {
                        this->time += dt * time_multiplier;
                        this->simulation_time.setString("time = " + std::to_string(this-
>time));
                }

                for (Particle& p : particles) {
                        sf::Vector2f posn = p.get_position();
                        if (!(posn.x > 800.f || posn.y > 600.f || posn.x < -10.f ||
posn.y < -10.f)) {
                                p.update_position(this->time);
                        }
                }
                this->draw(1.f / 60.f);
        }

}

void Plot::draw(float dt) {
        this->_data->window.clear(sf::Color(32, 32, 32));
        this->_data->window.draw(state_title);
        this->menu_button.draw(this->_data->window);

        if (current_input <= 4) {
                // hide UI elements when finished inputting
                this->function_entry.draw(this->_data->window);
                this->_data->window.draw(prompt);
        }

        if (finished_integrating) {
                for (Particle& p : particles) {
                        p.draw(this->_data->window);
                }

                this->_data->window.draw(simulation_time);
                this->save_button.draw(this->_data->window);
                if (this->typing_name) {
                        this->naming.draw(this->_data->window);
                }
        }

        this->_data->window.display();
}
```

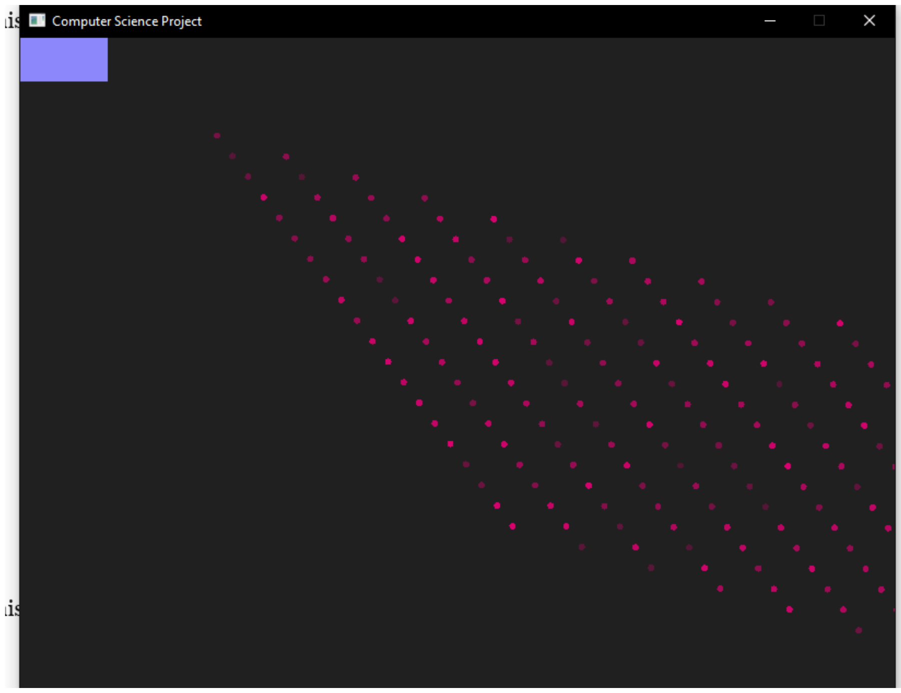
For testing purposes, I changed Menu.cpp, so that it calls the Plot(ProgramDataRef, Field) overload for the plot constructor when the menu is clicked.

```
case sf::Event::MouseButtonReleased:
                        if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
                                if (plot_button.is_mouse_over(this->_data->window)) {
                                        std::cout << "Plot button clicked" << std::endl;
                                        this->_data->state_machine.push_state(state_ref(new
Plot(_data, Field("my new vector field"))));
```

This is the same vector field that was saved a few pages ago.

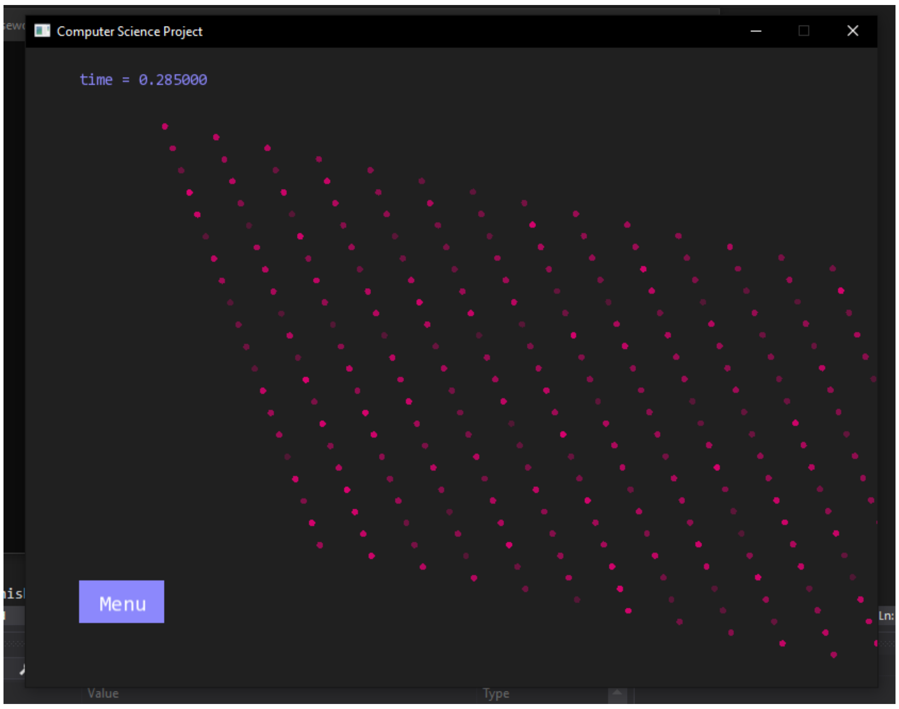


This is what I see immediately after clicking the menu button. After a while, it finishes integration and indeed I see this:



The test is half-success half-failure because it “worked” but the UI is messed up; indubitably I will fix this before too long.

I fixed the UI, the changes were trivial, so I won't paste them here.



```
time = 0.285000
```

I decided to omit the save button for Field-loaded simulations because if the vector field had been loaded from the file then it has already been saved.

Now all that's left to do is to implement the List state.

List.hpp

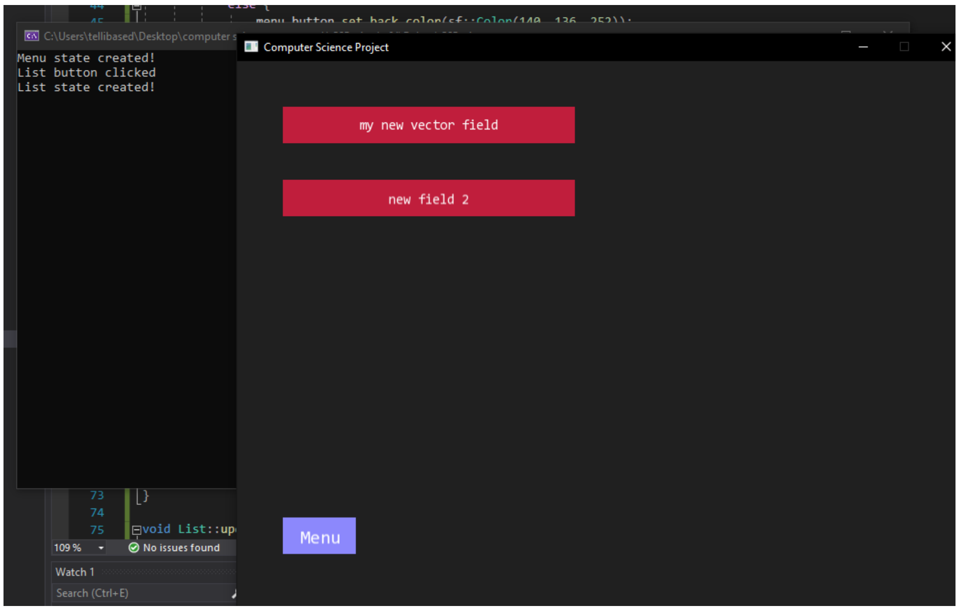
```
#pragma once

#include "Program.hpp"
#include "RK4.hpp"
#include "Particle.hpp"
#include "Button.hpp"
#include "Field.hpp"

class List : public State {
public:
	List(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	std::vector<Button> buttons;
	Button menu_button = Button("Menu", { 80, 40 }, 20, sf::Color(140, 136, 252),
sf::Color::White);
};
```

List.cpp

```
#include "List.hpp"
#include "Menu.hpp"

List::List(ProgramDataRef data) {
	this->_data = data;

	std::vector<std::string> names = get_names();

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });

	int i = 0;
	for (std::string s : names) {
		Button new_button = Button(s, { 320, 40 }, 15, sf::Color(192, 30, 60), 
sf::Color::White);
		new_button.set_font(this->_data->asset_manager.get_font("consolas"));
		new_button.set_position({ 50.f, 50.f + 80.f * i });
		this->buttons.push_back(new_button);

		i++;
	};
}

void List::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user 
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();

		case sf::Event::MouseMoved: {
			for (Button& b : buttons) {
				if (b.is_mouse_over(this->_data->window)) {
					b.set_back_color(sf::Color(222, 60, 90)); // change 
color if hovered over
				}
				else {
					b.set_back_color(sf::Color(192, 30, 60)); // else 
change back
				}
			}

			if (menu_button.is_mouse_over(this->_data->window)) {
				menu_button.set_back_color(sf::Color(110, 106, 222));
			}
			else {
				menu_button.set_back_color(sf::Color(140, 136, 252));
			}
		}

		case sf::Event::MouseButtonReleased:
			if (event.mouseButton.button == sf::Mouse::Left) { // left button 
released
				if (menu_button.is_mouse_over(this->_data->window)) {
					std::cout << "Menu button clicked" << std::endl;
					this->_data->state_machine.push_state(state_ref(new 
Menu(_data)));
				}
```

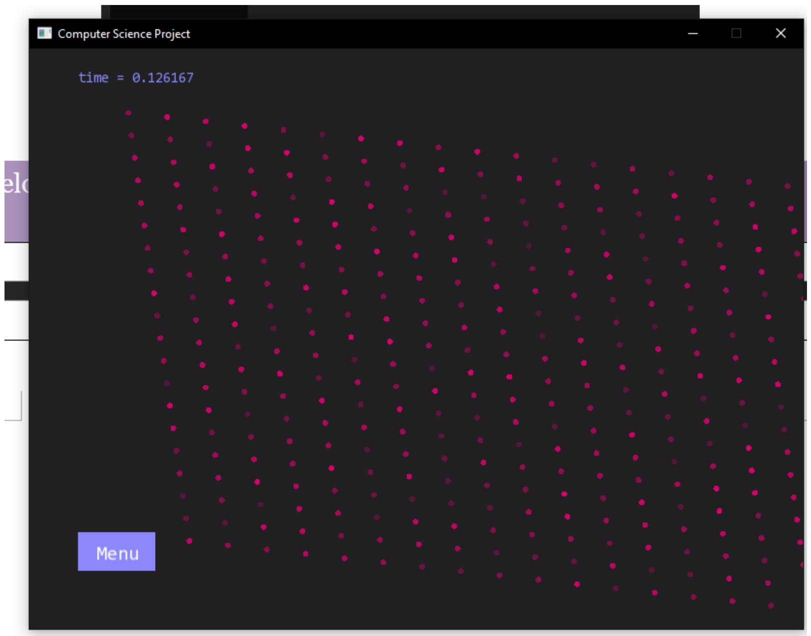
```
                                        for (Button& b : buttons) {
                                                if (b.is_mouse_over(this->_data->window)) {
                                                        std::cout <<
b.text.getString().toAnsiString() << std::endl;
                                                        this->_data-
>state_machine.push_state(state_ref(new Plot(_data,
b.text.getString().toAnsiString())));
                                                }
                                        }
                                }

                        }
                }
}

void List::init() {
        std::cout << "List state created!" << std::endl;

        //this->_data->asset_manager.load_texture("bg_menu", "bg_menu.PNG");
        //_background.setTexture(this->_data->asset_manager.get_texture("bg_menu"));
}

void List::update(float dt) {

}

void List::draw(float dt) {
        this->_data->window.clear(sf::Color(32, 32, 32));

        for (Button& b : buttons) {
                b.draw(this->_data->window);
        }

        this->menu_button.draw(this->_data->window);

        this->_data->window.display();
}
```

This is what it looks like with 2 vector fields in fields.csv



```
Menu state created!
List button clicked
List state created!
```

After clicking on the first one, I get:



```
time = 0.126167
```

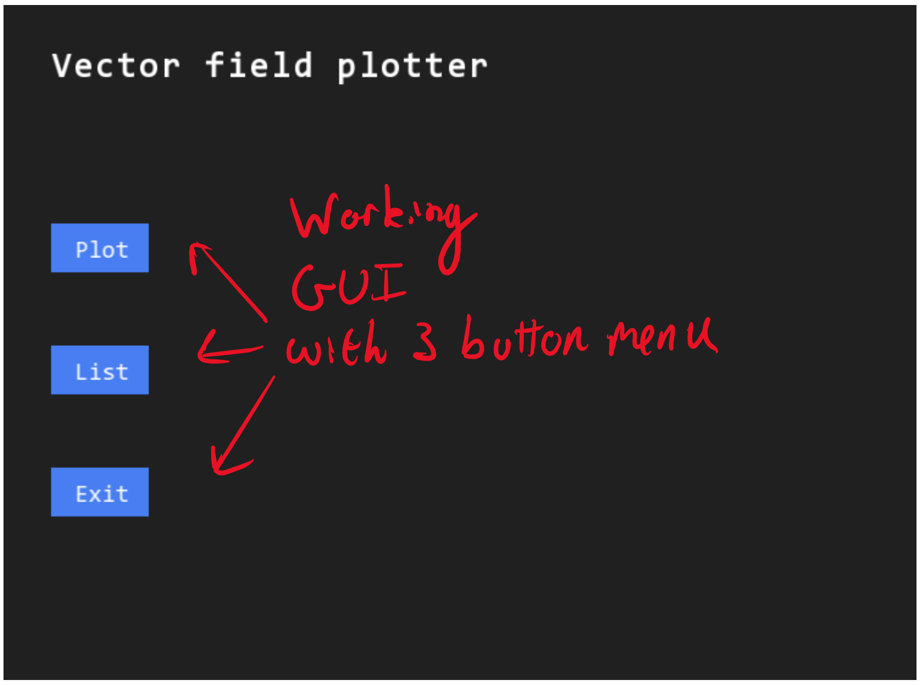
which is to be expected!

Since all features of the program are completed and have been tested I can say that development is complete, so I will go onto evaluation.

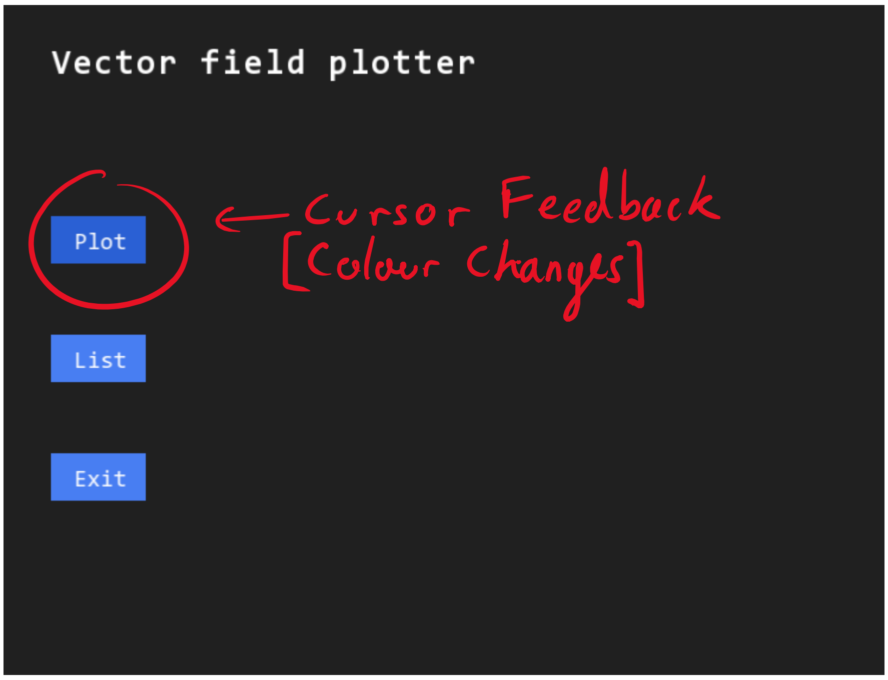
Evaluation

Testing to inform evaluation

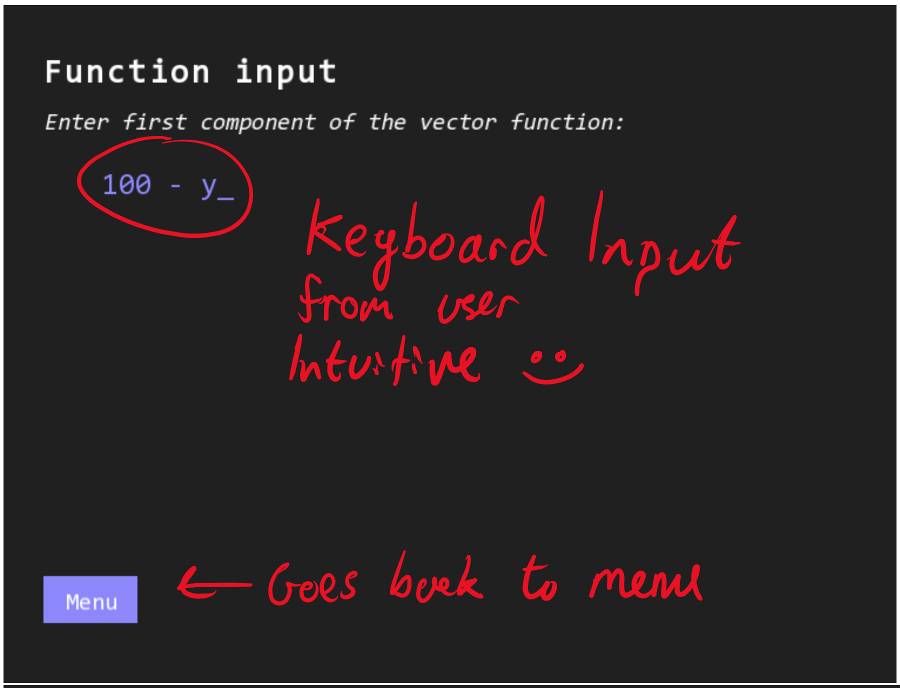
I will proceed to test for function & robustness of the finished program.



The GUI works perfectly well. The plot button replaces the Menu state with the Plot state when clicked, the List button with the List state, and the Exit button will exit out of the program.



When the user hovers their mouse over a button, the button colour changes; this helps with usability and maybe with accessibility for example if the user is using a small cursor or they have poor eyesight.



Function input

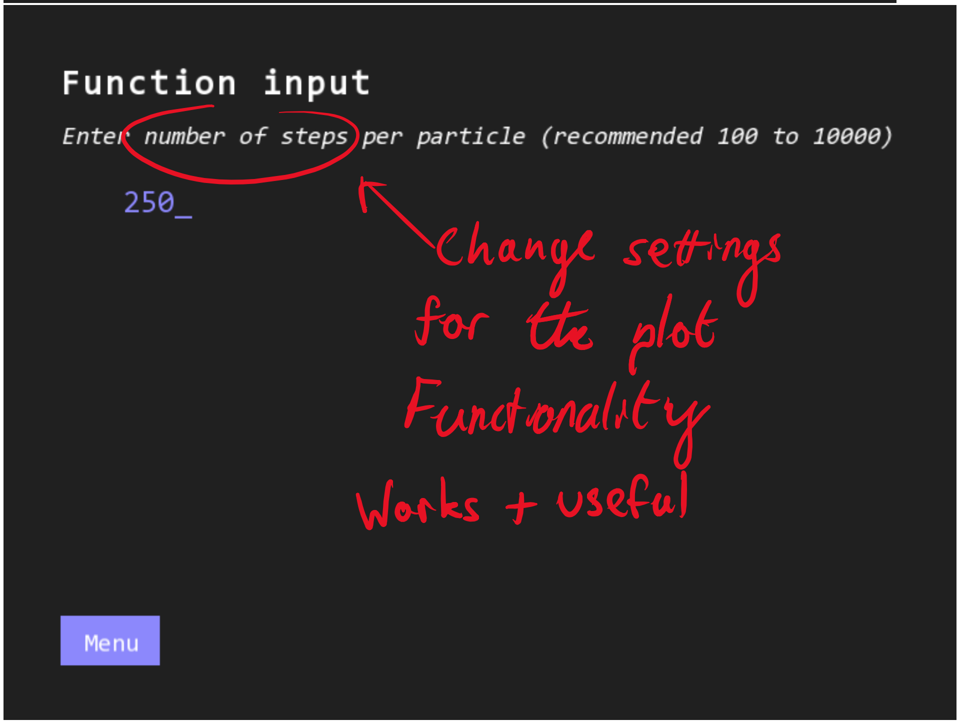
Enter first component of the vector function:

100 - y_

Keyboard Input from user Intuitive :)

← Goes back to menu

Usability and functionality feature: the input for the vector function is taken from the keyboard; not only is this intuitive for the user but is vital for the functionality of the program since the vector function is needed to solve the problem.



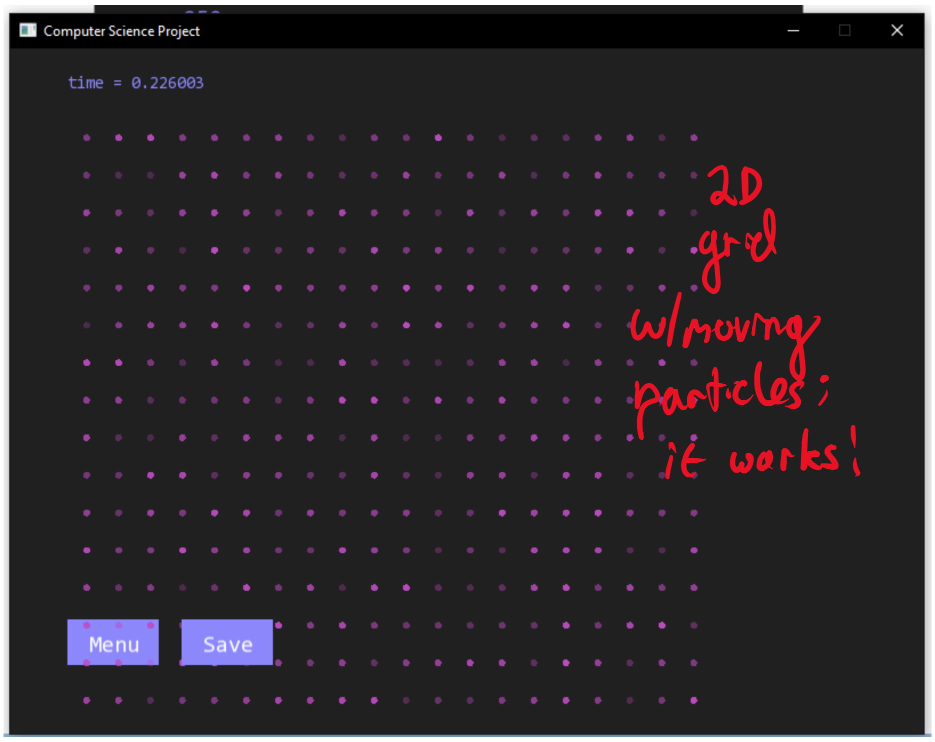
Function input

Enter number of steps per particle (recommended 100 to 10000)

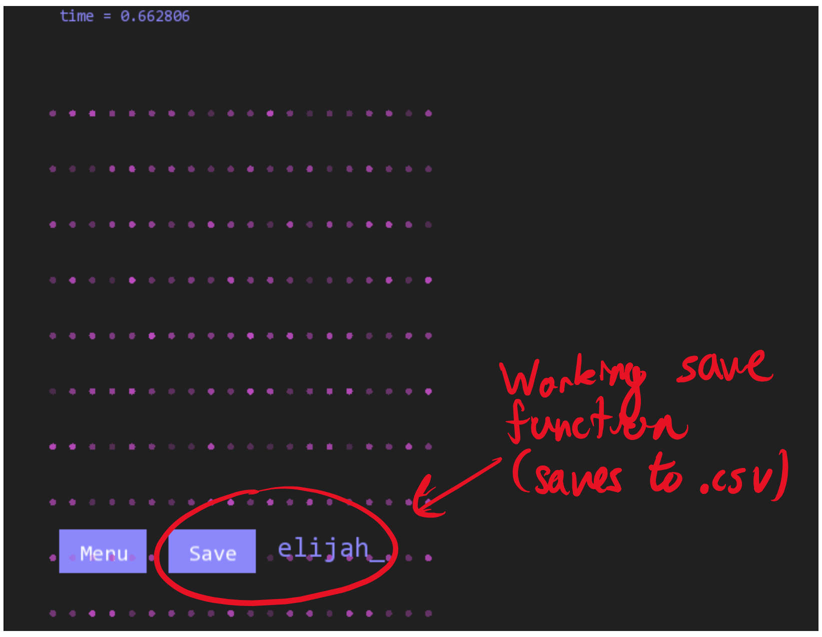
250_

Change settings for the plot Functionality Works + useful

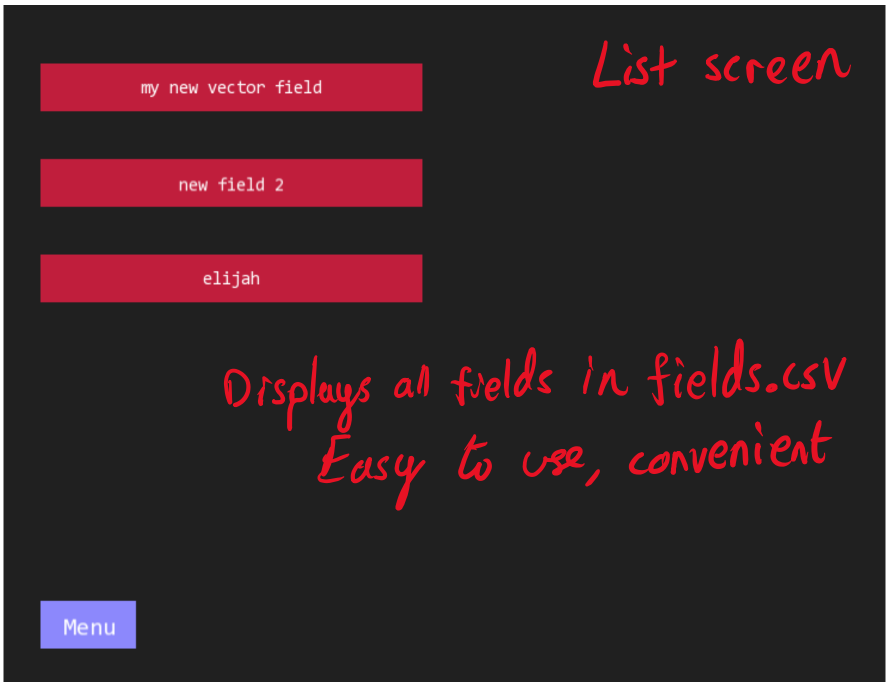
The user can input the number of steps and the end time of the simulation; this is a functionality feature to allow the user to customize their simulation for whatever they're doing. Its also useful for usability because users on weak PCs can input a lower number of steps to make the integration time shorter.

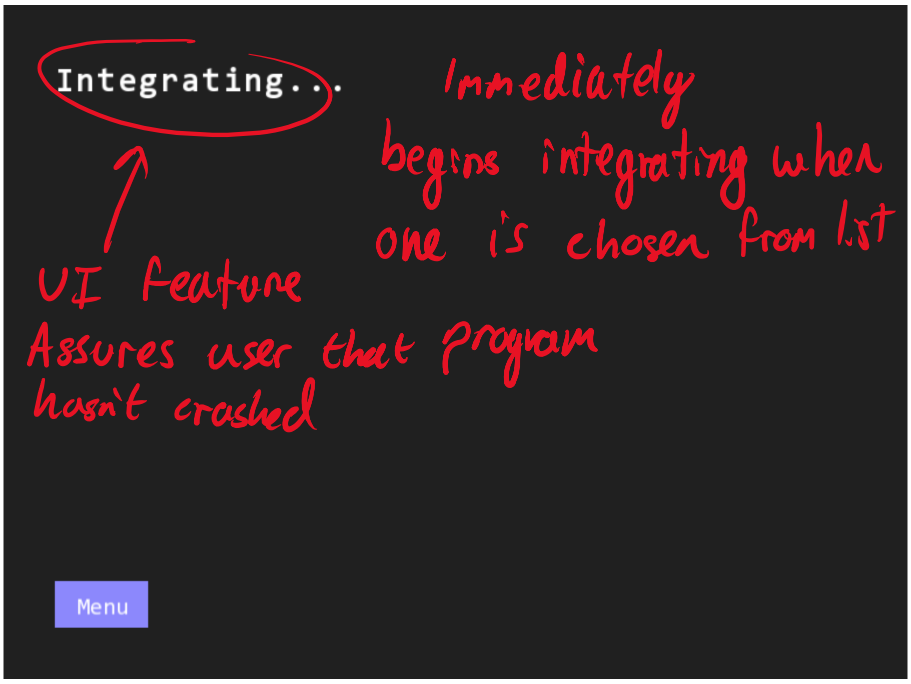


There is a 2D grid of particles which move according to the vector function that was input. Indubitably this is a completely vital piece of functionality to have in the program and if this were not implemented it wouldn't really be a program at all. The current time in the simulation is shown; this is done because it might be useful for the user, also to compare it with the end time that was input.



User can input a string of up to 32 characters and save their vector field (which includes the vector function, particle colour, number of steps and end time of simulation) inside a .csv file

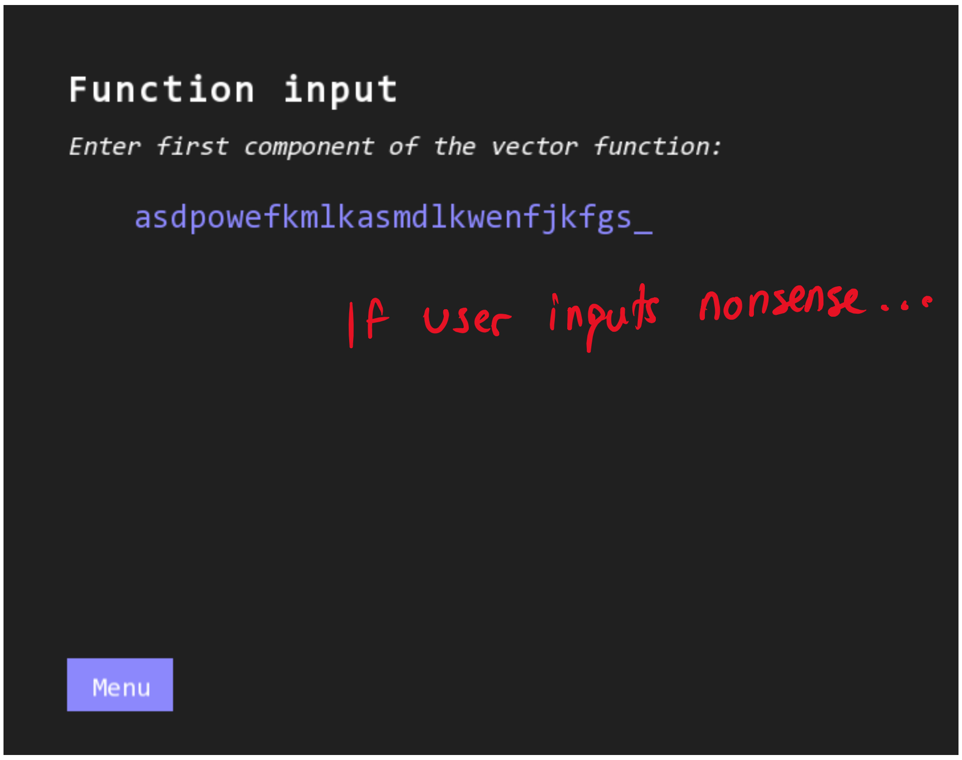



Integrating...
Immediately begins integrating when one is chosen from list
UI Feature
Assures user that program hasn't crashed
Menu

List state shows all of the vector fields stored in fields.csv; each field is given its own button, when the button is clicked the user goes to the plot state and immediately the program begins integrating the vector function. This is very high in usability, because if a user wants to re-use a vector field they can simply save it and use it later. All of the features that were shown up to and including this point are fully functional.

The "integrating…" message shown in the plot state is there because the program freezes for a few seconds during the RK4 integration process; this message assures the user that the program hasn't fully crashed but is simply busy.

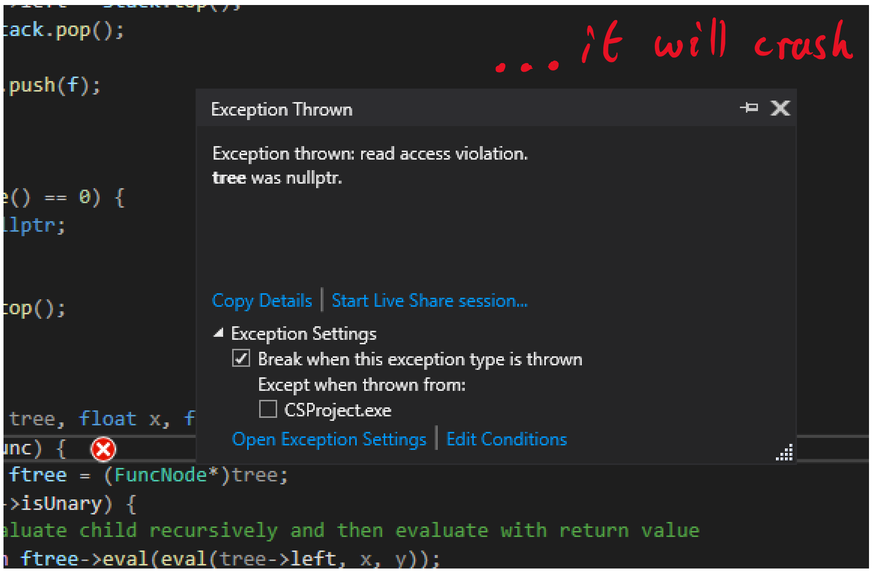
However..



Function input

Enter first component of the vector function:

```
asdpowefkmlkasmdlkwenfjkfgs_
```



```
ack.pop();

.push(f);

() == 0) {
llptr;

op();

 tree, float x, f
nc) {
 ftree = (FuncNode*)tree;
->isUnary) {
luate child recursively and then evaluate with return value
 ftree->eval(eval(tree->left, x, y));
```

Exception Thrown

Exception thrown: read access violation.
tree was nullptr.

...it will crash
Copy Details | Start Live Share session...
Exception Settings
☑ Break when this exception type is thrown
Except when thrown from:
☐ CSProject.exe
Open Exception Settings | Edit Conditions

Due to the lack of defensive design in the code, the program will crash given a malicious input. A similar error occurs when letters are put into the number of steps or the end time, etc.

Regardless, I have tested for robustness.

Referencing success criteria

I will address each success criterion in turn to see if they have been fully, partially or not met.

1. A main menu with buttons: Indubitably this criterion has been fully met; not only have I shown that a "main menu" exists (in the form of the Menu state) but that it also has 3 buttons, thereby fulfilling the criterion.
2. A list to browse vector fields from a database: This is quite controversial, because rather than using a database in the normal sense (i.e. a database managed by a relational database management system) I have actually gone and implemented a flat-file database (a .csv file). So I will say that this has been partially met; if I had more time I would have created a MySQL database and hosted it on a VPS but unfortunately I did not have that amount of time.

   In the analysis section, I spoke a fair bit about using the Internet for teachers to upload their vector field onto the database for their students, but with the flat-file approach taken in the final solution this is not possible. Regardless, the amount of information (function, steps, end time) that goes into plotting a vector field is relatively small and it's not like the teacher can't just tell the students that information directly.
3. A plotting screen: Yes, this criterion has been met. We have seen from the post-development testing that there is a plotting screen which contains particles moving according to the solution of the differential equation provided by the vector function. This criterion is therefore fully met.
4. User can input vector function: The criterion has been met; we have seen this from post-development testing; the user inputs the vector function using their keyboard.
5. Program takes in user keyboard input for vector function: This has been met, again see post-development testing, this is basically the same thing as criterion 4.
6. A grid with particles: Again, this has been met; in the plot state the particles are on a grid (a grid being an integer lattice i.e. the points inside the window and not necessarily a visible array of lines which intersect orthogonally, I made this distinction in the analysis)
7. Particles move according to vector function: Indubitably – I have implemented the Runge-Kutta order 4 algorithm which solves the differential equation thus allowing the particles to move according to the vector function, where its velocity at a given position is equal to the vector function input by the user. This can be seen from the evidence because the particles are moving and their path is determined from the vector function.
8. Ability to go back to the main menu: This has been met, we have seen this from the screenshots in post-development testing which show a "Menu" button in the states where it is relevant.
9. Ability to save vector field to a database: This has been partially met; we have seen this from the post-development testing screenshots where you can see the "Save" button which saves the vector field to a .csv flat-file database.
10. Ability to configure settings on vector field plot: This has been met although you could say partially; post-development testing screenshots show you can change the number of steps and end time of the simulation but not the speed of the plot.

There are no criteria which were completely unfulfilled but 3 of them were met partially. Two related criteria from these 3, the "list to browse from a database" and "ability to save vector field to database" are partial because I said I would use MySQL when I ended up using .csv. To amend this in further development, I would set up a VPS and host a MySQL database on it, and send requests to the database instead of reading and writing from a .csv file. This would increase the usability and convenience of the program since people could now share what they have made on the Internet.

The third one which is partial is the "ability to configure settings on vector field plot", because not all settings can be changed. If I wanted to amend this in further development, I would include another keyboard input for the time multiplier (which changes the speed of the plot) and perhaps another input for the colour of the particles (although I made it so that, unless loaded from the .csv, the colour is random each time which I prefer)

Usability

The fact that the program uses GUI over CLI (pg. 1-5) itself acts as a usability feature, because if the program weren't a GUI program then it wouldn't be able to solve the program it's trying to solve (plotting a vector field).

The prompts on the screen (pg. 2) "Enter first component of vector function", "Enter second component of vector function", "Enter number of steps", etc. act as another usability feature; without these the user wouldn't know what to input, in fact they probably wouldn't know they had to input anything in the first place. Therefore these usability features are vital to the functionality of the whole program, they need to be there lest the user becomes extremely confused.

The buttons on the screen (pg. 1) help with usability because it makes it very easy for the user to go between states; if there were no buttons the whole flow of the program would become pretty difficult to navigate especially with a GUI program.

The "Integrating…" message (pg. 4) acts as a useful usability feature; without it the user might think that the program has crashed. The RK4 integrator can take a long time if you put a high number of steps and/or a high end time. If the user thinks the program has crashed while it's actually busy working then the user might forcibly close the program and never see the plot screen. This message reassures the user that the program is just working and hasn't crashed. However, I think this is only a partially successful feature; if I wanted to build on it more in future development I would add a progress bar at the top during integration although this would be hard to add because you would have to estimate how long the integration would last and how far through it's completed.

An unmet usability feature is that of defensive design; currently if the user accidently types something that is parsed as incorrect by the shunting-yard algorithm the program will completely crash (pg. 5). If I had more time, I would add validation to the user input to ensure they aren't passing in nonsense to crash the program, because if the user makes a small typo currently, the program will crash, which is very bad for usability.

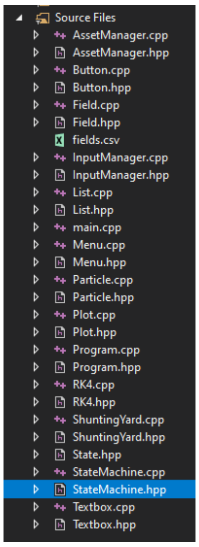
Limitations

One maintenance issue is that the code isn't fully commented; this is a problem because if someone else were to look at the code they might not understand what is happening or how the different files tie together. To solve this I would simply add more comments to the code if I had time. Example from the RK4 integrator:

```
void Integrator::integrate() {
        double h = end_time / steps;

        for (int step = 0; step < steps; step++) {
                sf::Vector2f point_1 = this->eval_vector_function(solution[step * h]);
                sf::Vector2f point_2 = this->eval_vector_function(solution[step * h] +
((float)h/2)*point_1);
                sf::Vector2f point_3 = this->eval_vector_function(solution[step * h] +
((float)h/2)*point_2);
                sf::Vector2f point_4 = this->eval_vector_function(solution[step * h] +
(float)h*point_3);

                solution[h * (step + 1.f)] = solution[h * step] + ((float)h / 6) *
(point_1 + 2.f * point_2 + 2.f * point_3 + point_4);
        }
}
```

It is not obvious what this code is doing, nor why or how, so clearly comments are needed.

Speaking of RK4 integrator, a limitation of the solution is that the integrator usually takes a relatively long amount of time to finish. If I were to approach a problem like this again, I would have to look into ways of optimising the RK4 algorithm, probably with multithreading/SIMD processing. This became a real problem because I had to start limiting the number of particles I was using in order to cut down the integration time.



```
Source Files
    AssetManager.cpp
    AssetManager.hpp
    Button.cpp
    Button.hpp
    Field.cpp
    Field.hpp
    fields.csv
    InputManager.cpp
    InputManager.hpp
    List.cpp
    List.hpp
    main.cpp
    Menu.cpp
    Menu.hpp
    Particle.cpp
    Particle.hpp
    Plot.cpp
    Plot.hpp
    Program.cpp
    Program.hpp
    RK4.cpp
    RK4.hpp
    ShuntingYard.cpp
    ShuntingYard.hpp
    State.hpp
    StateMachine.cpp
    StateMachine.hpp
    Textbox.cpp
    Textbox.hpp
```

Another maintenance issue is due to the sheer size of the program; it spans 28 source files (though most of the source files are hpp-cpp pairs) and therefore I feel it could be very difficult to tie everything together. If I wanted to work on this project with another person, or if I came back to it in a year, I would have to see some sort of entity-relationship diagram. In spite of this, the code is fairly modular and well-organised (in my opinion.)

Speaking of entity-relationship diagrams, I had to deviate quite heavily from what I determined in the UML relationship diagram shown in the design section. I had to do this simply because I didn't know what I was in store for before I started coding the solution; there were many new classes I had to introduce that I didn't know I would need, for example the Integrator class, the Particle and Field classes, etc.

Another (debatable) limitation is that the program isn't very nice to look at; if I had more time I would like to add visual effects on the Plot screen although I realise this won't be the most economical advancement for school computers. I would probably use shaders to add effects.

Regarding school computers, I realise another limitation is that the program isn't very easy to use, in my opinion. This is because of the technical jargon "number of steps per particle", "end time", etc. This is a problem in school environments, because if students want to just learn about vector fields then this technical jargon just serves as another hurdle to get over which is detrimental to the teacher.

AssetManager.hpp

```cpp
#pragma once

#include <map>
#include <SFML/Graphics.hpp>

class AssetManager {
public:
	void load_texture(std::string name, std::string filename);
	sf::Texture& get_texture(std::string name);

	void load_font(std::string name, std::string filename);
	sf::Font& get_font(std::string name);

private:
	std::map<std::string, sf::Texture> _textures;
	std::map<std::string, sf::Font> _fonts;
};
```

AssetManager.cpp

```cpp
#include "AssetManager.hpp"

void AssetManager::load_texture(std::string name, std::string filename) {
	sf::Texture texture;

	if (texture.loadFromFile(filename)) {
		this->_textures[name] = texture;
	}
}

sf::Texture& AssetManager::get_texture(std::string name) {
	return this->_textures.at(name);
}

void AssetManager::load_font(std::string name, std::string filename) {
	sf::Font font;

	if (font.loadFromFile(filename)) {
		this->_fonts[name] = font;
	}
}

sf::Font& AssetManager::get_font(std::string name) {
	return this->_fonts.at(name);
}
```

Button.hpp
```cpp
#pragma once

#include <iostream>
#include <SFML/Graphics.hpp>

class Button {
public:
	Button(std::string button_text, sf::Vector2f size, int char_size, sf::Color
bg_color, sf::Color text_color);
	void set_font(sf::Font& font);
	void set_back_color(sf::Color color);
	void set_text_color(sf::Color color);
	void set_position(sf::Vector2f pos);
	void draw(sf::RenderWindow& window);
	bool is_mouse_over(sf::RenderWindow& window);
	sf::Text text;
private:
	sf::RectangleShape button;
};
```

Button.cpp

```cpp
#include "Button.hpp"

Button::Button(std::string button_text, sf::Vector2f size, int char_size, sf::Color
bg_color, sf::Color text_color) {
	this->text.setString(button_text);
	this->text.setFillColor(text_color);
	this->text.setCharacterSize(char_size);

	this->button.setSize(size);
	this->button.setFillColor(bg_color);
}

void Button::set_font(sf::Font& font) {
	this->text.setFont(font);
}

void Button::set_back_color(sf::Color color) {
	this->button.setFillColor(color);
}

void Button::set_text_color(sf::Color color) {
	this->text.setFillColor(color);
}

void Button::set_position(sf::Vector2f pos) {
	this->button.setPosition(pos);

	float x_pos = pos.x + (button.getLocalBounds().width -
text.getLocalBounds().width) / 2;
	float y_pos = pos.y + (button.getLocalBounds().height -
text.getLocalBounds().height) / 2;

	this->text.setPosition({ x_pos, y_pos - text.getCharacterSize()/4});
}

void Button::draw(sf::RenderWindow& window) {
	window.draw(this->button);
	window.draw(this->text);
}
```

```
bool Button::is_mouse_over(sf::RenderWindow& window) {
	float mouse_x = sf::Mouse::getPosition(window).x;
	float mouse_y = sf::Mouse::getPosition(window).y;

	float button_x = this->button.getPosition().x;
	float button_y = this->button.getPosition().y;

	float button_pos_width = this->button.getPosition().x + this-
>button.getLocalBounds().width;
	float button_pos_height= this->button.getPosition().y + this-
>button.getLocalBounds().height;

	return (mouse_x < button_pos_width && mouse_x > button_x && mouse_y <
button_pos_height && mouse_y > button_y);
}
```

Field.hpp

```
#include "ShuntingYard.hpp"
#pragma once

#include <SFML/Graphics.hpp>

class Field {
public:
	Field(std::string x_component, std::string y_component, int steps, float
end_time, sf::Color color);
	Field(std::string name);
	void write(std::string name); // writes to
	std::string x_component;
	std::string y_component;
	int steps;
	float end_time;
	sf::Color color;
};

int find_name(std::string name);
std::vector<std::string> split_by_semi(std::string str);
std::vector<std::string> get_names();
```

Field.cpp

```cpp
#include "Field.hpp"
#include <fstream>
#include <iostream>

std::vector<std::string> get_names() {
	std::vector<std::string> result;

	std::ifstream fields_file;
	std::string line;

	fields_file.open("fields.csv");
	while (getline(fields_file, line)) {
		std::string next_name;

		for (char c : line) {
			if (c != ';') {
				next_name += c;
			}
			else {
				result.push_back(next_name);
				break;
			}
		}
	}

	return result;
}

//helper
int find_name(std::string name) {
	std::vector<std::string> names = get_names();

	for (int i = 0; i < names.size(); i++) {
		if (names[i] == name) {
			return i;
		}
	}

	return -1; // not found
}

std::vector<std::string> split_by_semi(std::string str) {
	std::vector<std::string> result;
	std::string next_string;

	for (char c : str) {
		if (c == ';') {
			result.push_back(next_string);
			next_string = "";
		}
		else {
			next_string += c;
		}
	}

	result.push_back(next_string);

	return result;
}
```

```
Field::Field(std::string x_component, std::string y_component, int steps, float
end_time, sf::Color color) {
	this->x_component = x_component;
	this->y_component = y_component;
	this->steps = steps;
	this->end_time = end_time;
	this->color = color;
}

Field::Field(std::string name) {
	int index = find_name(name);

	std::cout << index << std::endl;

	std::ifstream fields_file;
	std::string line;

	int i = 0;
	fields_file.open("fields.csv");
	while (getline(fields_file, line)) {
		if (i == index) {
			break;
		}
	}

	std::vector<std::string> split = split_by_semi(line);

	this->x_component = split[1];
	this->y_component = split[2];
	this->steps = stoi(split[3]);
	this->end_time = stof(split[4]);
	this->color = sf::Color(stoi(split[5]), stoi(split[6]), stoi(split[7]));
}



void Field::write(std::string name) {
	std::vector<std::string> names = get_names();

	std::ofstream fields_file;
	fields_file.open("fields.csv", std::ios_base::app);

	fields_file << name << ";" << x_component << ";" << y_component
		<< ";" << steps << ";" << end_time << ";" << (int)color.r
		<< ";" << (int)color.g << ";" << (int)color.b << "\n";

	fields_file.close();
}
```

fields.csv

```
my new vector field;x + y;x - y;100;1;216;1;110
new field 2;(-1) * y;x;100;1;22;244;37
elijah;(-1) * x;y;100;1;187;75;191
```

InputManager.hpp

```
#pragma once

#include <SFML/Graphics.hpp>

class InputManager {
public:
	InputManager() {};
	~InputManager() {};

	bool is_sprite_clicked(sf::Sprite object, sf::Mouse::Button button, 
sf::RenderWindow& window);
	sf::Vector2i get_mouse_position(sf::RenderWindow& window);
};
```

InputManager.cpp

```
#include "InputManager.hpp"

bool InputManager::is_sprite_clicked(sf::Sprite object, sf::Mouse::Button button, 
sf::RenderWindow& window) {
	if (sf::Mouse::isButtonPressed(button)) {
		sf::IntRect rect(object.getPosition().x, object.getPosition().y, 
object.getGlobalBounds().width, object.getGlobalBounds().height);

		return rect.contains(sf::Mouse::getPosition(window));
	}

	return false;
}

sf::Vector2i InputManager::get_mouse_position(sf::RenderWindow& window) {
	return sf::Mouse::getPosition(window);
}
```

List.hpp

```
#pragma once

#include "Program.hpp"
#include "RK4.hpp"
#include "Particle.hpp"
#include "Button.hpp"
#include "Field.hpp"

class List : public State {
public:
	List(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	std::vector<Button> buttons;
	Button menu_button = Button("Menu", { 80, 40 }, 20, sf::Color(140, 136, 252), 
sf::Color::White);
};
```

List.cpp
```
#include "List.hpp"
#include "Menu.hpp"

List::List(ProgramDataRef data) {
	this->_data = data;

	std::vector<std::string> names = get_names();

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });

	int i = 0;
	for (std::string s : names) {
		Button new_button = Button(s, { 320, 40 }, 15, sf::Color(192, 30, 60),
sf::Color::White);
		new_button.set_font(this->_data->asset_manager.get_font("consolas"));
		new_button.set_position({ 50.f, 50.f + 80.f * i });
		this->buttons.push_back(new_button);

		i++;
	};
}

void List::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();

		case sf::Event::MouseMoved: {
			for (Button& b : buttons) {
				if (b.is_mouse_over(this->_data->window)) {
					b.set_back_color(sf::Color(222, 60, 90)); // change
color if hovered over
				}
				else {
					b.set_back_color(sf::Color(192, 30, 60)); // else
change back
				}
			}

			if (menu_button.is_mouse_over(this->_data->window)) {
				menu_button.set_back_color(sf::Color(110, 106, 222));
			}
			else {
				menu_button.set_back_color(sf::Color(140, 136, 252));
			}
		}

		case sf::Event::MouseButtonReleased:
			if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
				if (menu_button.is_mouse_over(this->_data->window)) {
					std::cout << "Menu button clicked" << std::endl;
					this->_data->state_machine.push_state(state_ref(new
Menu(_data)));
				}
```

```
                            for (Button& b : buttons) {
                                   if (b.is_mouse_over(this->_data->window)) {
                                          std::cout <<
b.text.getString().toAnsiString() << std::endl;
                                          this->_data-
>state_machine.push_state(state_ref(new Plot(_data,
b.text.getString().toAnsiString())));
                                   }
                            }
                     }

              }
       }
}

void List::init() {
       std::cout << "List state created!" << std::endl;

       //this->_data->asset_manager.load_texture("bg_menu", "bg_menu.PNG");
       //_background.setTexture(this->_data->asset_manager.get_texture("bg_menu"));
}

void List::update(float dt) {

}

void List::draw(float dt) {
       this->_data->window.clear(sf::Color(32, 32, 32));

       for (Button& b : buttons) {
              b.draw(this->_data->window);
       }

       this->menu_button.draw(this->_data->window);

       this->_data->window.display();
}
```

main.cpp

```
#include <SFML/Graphics.hpp>
#include "Program.hpp"
#include "ShuntingYard.hpp"
#include <iostream>

int main()
{

    Program program = Program(800, 600, "Computer Science Project");

    return 0;
}
```

Menu.hpp

```
#pragma once

#include "State.hpp"
#include "Button.hpp"
#include "Plot.hpp"
#include "Program.hpp"

class Menu : public State {
public:
	Menu(ProgramDataRef data);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);
private:
	ProgramDataRef _data;
	sf::Clock _clock;
	sf::Sprite _background;
	Button plot_button = Button("Plot", { 80, 40 }, 20, sf::Color(72, 126, 242),
sf::Color::White);
	Button list_button = Button("List", { 80, 40 }, 20, sf::Color(72, 126, 242),
sf::Color::White);
	Button exit_button = Button("Exit", { 80, 40 }, 20, sf::Color(72, 126, 242),
sf::Color::White);
	sf::Text title;
};
```

Menu.cpp

```
#include "Menu.hpp"
#include "Field.hpp"
#include "List.hpp"

Menu::Menu(ProgramDataRef data) {
	this->_data = data;

	this->_data->asset_manager.load_font("consolas", "consola.ttf");
	this->plot_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->list_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->exit_button.set_font(this->_data->asset_manager.get_font("consolas"));

	this->title.setFont(this->_data->asset_manager.get_font("consolas"));
	this->title.setPosition({ 50, 50 });
	this->title.setCharacterSize(30);
	this->title.setString("Vector field plotter");
	this->title.setStyle(sf::Text::Style::Bold);

	this->plot_button.set_position({ 50, 200 });
	this->list_button.set_position({ 50, 300 });
	this->exit_button.set_position({ 50, 400 });
}

void Menu::init() {
	std::cout << "Menu state created!" << std::endl;

	//this->_data->asset_manager.load_texture("bg_menu", "bg_menu.PNG");
	//_background.setTexture(this->_data->asset_manager.get_texture("bg_menu"));
}
```

```
void Menu::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();

		case sf::Event::MouseMoved:
			for (Button* b : {&plot_button, &list_button, &exit_button}) { //
iterate over references to buttons
				if (b->is_mouse_over(this->_data->window)) {
					b->set_back_color(sf::Color(42, 96, 212)); //
change color if hovered over
				} else {
					b->set_back_color(sf::Color(72, 126, 242)); // else
change back
				}
			}

		case sf::Event::MouseButtonReleased:
			if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
				if (plot_button.is_mouse_over(this->_data->window)) {
					std::cout << "Plot button clicked" << std::endl;
					this->_data->state_machine.push_state(state_ref(new
Plot(_data)));
				} else if(list_button.is_mouse_over(this->_data->window))
{

					std::cout << "List button clicked" << std::endl;
					this->_data->state_machine.push_state(state_ref(new
List(_data)));
				} else if(exit_button.is_mouse_over(this->_data->window))
{

					std::cout << "Exit button clicked" << std::endl;
					this->_data->window.close();
				}
			}

		}
	}
}

void Menu::update(float dt) {

}

void Menu::draw(float dt) {
	this->_data->window.clear(sf::Color(32,32,32));
	this->_data->window.draw(this->_background);
	this->_data->window.draw(this->title);
	this->plot_button.draw(this->_data->window);
	this->list_button.draw(this->_data->window);
	this->exit_button.draw(this->_data->window);
	this->_data->window.display();
}
```

Particle.hpp

```
#pragma once

#include "RK4.hpp"
#include <SFML/Graphics.hpp>

class Particle {
public:
	Particle(sf::Color color, float radius, int steps, int end_time, sf::Vector2f
initial_position,
		ShuntingYard::Node* x_component, ShuntingYard::Node* y_component);
	void update_position(float time);
	void draw(sf::RenderWindow& window);
	sf::Vector2f get_position();
private:
	float radius;
	int steps;
	float end_time;
	sf::Vector2f initial_position;
	sf::CircleShape form;
	ShuntingYard::Node* x_component;
	ShuntingYard::Node* y_component;
	Integrator integrator = Integrator(steps, end_time, x_component, y_component,
initial_position);
};
```

Particle.cpp

```
#include "Particle.hpp"


Particle::Particle(sf::Color color, float radius, int steps, int end_time,
sf::Vector2f initial_position,
	ShuntingYard::Node* x_component, ShuntingYard::Node* y_component) {
	this->form.setFillColor(color);
	this->form.setRadius(radius);
	this->initial_position = initial_position;
	this->form.setPosition(initial_position);
	this->x_component = x_component;
	this->y_component = y_component;
	this->steps = steps;
	this->end_time = end_time;

	this->integrator = Integrator(steps, end_time, x_component, y_component,
initial_position);
	this->integrator.integrate();
}

void Particle::update_position(float time) {
	this->form.setPosition(this->integrator.get_position(time));
}

void Particle::draw(sf::RenderWindow& window) {
	window.draw(this->form);
}

sf::Vector2f Particle::get_position() {
	return this->form.getPosition();
}
```

Plot.hpp

```
#pragma once

#include "Textbox.hpp"
#include "Program.hpp"
#include "RK4.hpp"
#include "Particle.hpp"
#include "Button.hpp"
#include "Field.hpp"

class Plot : public State {
public:
	Plot(ProgramDataRef data);
	Plot(ProgramDataRef data, Field basis);

	void init();
	void handle_input();
	void update(float dt);
	void draw(float dt);

	void integrate();
private:
	ProgramDataRef _data;
	float time_multiplier = 0.01;
	float time = 0;
	float end_time = 0;
	int steps = 0;
	int current_input = 1;
	bool finished_integrating = false;
	bool typing_name = false;
	bool created_from_field = false;
	std::string x_text;
	std::string y_text;
	std::vector<Particle> particles;
	sf::Color particle_color;
	ShuntingYard::Node* x_component;
	ShuntingYard::Node* y_component;
	Textbox function_entry = Textbox(25, sf::Color(140, 136, 252), true);
	Textbox naming = Textbox(25, sf::Color(140, 136, 252), false);
	Button menu_button = Button("Menu", { 80, 40 }, 20, sf::Color(140, 136, 252), 
sf::Color::White);
	Button save_button = Button("Save", { 80, 40 }, 20, sf::Color(140, 136, 252), 
sf::Color::White);
	sf::Text prompt;
	sf::Text state_title;
	sf::Text simulation_time;
};
```

Plot.cpp

```
#include "Plot.hpp"
#include <iostream>
#include <random>
#include "Menu.hpp"

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis1(64, 255);
std::uniform_int_distribution<> dis2(0, 255);

Plot::Plot(ProgramDataRef data) {
	this->_data = data;
	this->particle_color = sf::Color(dis2(gen), dis2(gen), dis2(gen));

	function_entry.set_font(this->_data->asset_manager.get_font("consolas"));
	function_entry.set_position({ 100, 150 });

	naming.set_font(this->_data->asset_manager.get_font("consolas"));
	naming.set_position({ 250, 500 });
	naming.set_limit(32);

	prompt.setFont(this->_data->asset_manager.get_font("consolas"));
	prompt.setString("Enter first component of the vector function:");
	prompt.setCharacterSize(20);
	prompt.setPosition({ 50, 100 });
	prompt.setStyle(sf::Text::Italic);

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setString("Function input");
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);

	simulation_time.setFont(this->_data->asset_manager.get_font("consolas"));
	simulation_time.setCharacterSize(15);
	simulation_time.setPosition({ 50, 20 });
	simulation_time.setFillColor(sf::Color(140, 136, 252));

	save_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->save_button.set_position({ 150, 500 });

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });
}

Plot::Plot(ProgramDataRef data, Field field) {
	this->created_from_field = true;

	this->_data = data;
	this->particle_color = field.color;
	this->end_time = field.end_time;
	this->steps = field.steps;
	this->current_input = 5;

	std::cout << field.x_component << std::endl;
	std::cout << field.y_component << std::endl;

	this->x_component = 
ShuntingYard::parse(ShuntingYard::reversePolishNotation(field.x_component.c_str()));
```

```
	this->y_component =
ShuntingYard::parse(ShuntingYard::reversePolishNotation(field.y_component.c_str()));

	simulation_time.setFont(this->_data->asset_manager.get_font("consolas"));
	simulation_time.setCharacterSize(15);
	simulation_time.setPosition({ 50, 20 });
	simulation_time.setFillColor(sf::Color(140, 136, 252));

	state_title.setFont(this->_data->asset_manager.get_font("consolas"));
	state_title.setCharacterSize(30);
	state_title.setPosition({ 50, 50 });
	state_title.setStyle(sf::Text::Bold);

	this->menu_button.set_font(this->_data->asset_manager.get_font("consolas"));
	this->menu_button.set_position({ 50, 500 });

	integrate();
}

void Plot::integrate() {
	this->state_title.setString("Integrating...");
	this->draw(0); // show "Integrating" message before initialising particles
	// if this weren't here it would run the below block of code before re-drawing

	// create 2D grid of particles
	for (int i = 0; i < 20; i++) {
		for (int j = 0; j < 20; j++) {
			Particle new_particle = Particle(sf::Color(particle_color.r,
particle_color.g, particle_color.b, dis1(gen)),
													3.f,
steps, end_time, { 80.f + 35.1f * i, 60.f + 26.2f * j },
				this->x_component, this->y_component);

			new_particle.update_position(0);
			this->particles.push_back(new_particle);

		}
	}

	this->state_title.setString("");
	this->finished_integrating = true;
	this->time = 0;
}

void Plot::init() {
	std::cout << "Plot state created!" << std::endl;
}

void Plot::handle_input() {
	sf::Event event;

	while (this->_data->window.pollEvent(event)) { // get events that happen (user
inputs etc)
		switch (event.type) {
		case sf::Event::Closed: // clicks x in top right
			this->_data->window.close();
		case sf::Event::TextEntered:
			if (event.text.unicode != ENTER) {
				function_entry.on_type(event);
				naming.on_type(event);
			} else {
				if (current_input == 1) {
```

```
                                        prompt.setString("Enter second component of vector
function:");

                                        x_text = function_entry.get_text();

                                        ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(x_text.c_str());
                                        this->x_component = ShuntingYard::parse(rpn);

                                        this->function_entry.set_text("");

                                        current_input++;
                                } else if (current_input == 2) {
                                        y_text = function_entry.get_text();

                                        ShuntingYard::RPN rpn =
ShuntingYard::reversePolishNotation(y_text.c_str());
                                        this->y_component = ShuntingYard::parse(rpn);

                                        this->function_entry.set_text("");

                                        prompt.setString("Enter number of steps per
particle (recommended 100 to 10000)");

                                        current_input++;
                                } else if (current_input == 3) {
                                        steps = stoi(function_entry.get_text());

                                        prompt.setString("Enter end time");

                                        this->function_entry.set_text("");

                                        current_input++;

                                }
                                else if (current_input == 4) {
                                        end_time = stof(function_entry.get_text());

                                        function_entry.set_selected(false);
                                        current_input++;

                                        std::cout << end_time << std::endl;
                                        std::cout << steps << std::endl;

                                        integrate();
                                } else if (typing_name) {
                                        // add to file
                                        Field field = Field(x_text, y_text, steps,
end_time, sf::Color(particle_color.r, particle_color.g, particle_color.b));

                                        field.write(naming.get_text());
                                }
                        }
                case sf::Event::MouseMoved: // iterate over references to buttons
                {
                        for (Button* b : { &menu_button, &save_button }) {
                                if (b->is_mouse_over(this->_data->window)) {
                                        b->set_back_color(sf::Color(110, 106, 222));
                                } else {
                                        b->set_back_color(sf::Color(140, 136, 252));
                                }
                        }
```

```
			}


			case sf::Event::MouseButtonReleased:
				if (event.mouseButton.button == sf::Mouse::Left) { // left button
released
					if (menu_button.is_mouse_over(this->_data->window)) {
						std::cout << "Menu button clicked" << std::endl;
						this->_data->state_machine.push_state(state_ref(new
Menu(_data)));
					} else if (save_button.is_mouse_over(this->_data->window))
{

						std::cout << "Save button clicked" << std::endl;
						this->typing_name = true;
						this->naming.set_selected(true);
					}
				}

			}

		}
}

void Plot::update(float dt) {
	if (finished_integrating) {
		if (dt <= 1) {
			this->time += dt * time_multiplier;
			this->simulation_time.setString("time = " + std::to_string(this-
>time));
		}

		for (Particle& p : particles) {
			sf::Vector2f posn = p.get_position();
			if (!(posn.x > 800.f || posn.y > 600.f || posn.x < -10.f ||
posn.y < -10.f)) {
				p.update_position(this->time);
			}
		}
		this->draw(1.f / 60.f);
	}

}

void Plot::draw(float dt) {
	this->_data->window.clear(sf::Color(32, 32, 32));
	this->_data->window.draw(state_title);
	this->menu_button.draw(this->_data->window);

	if (current_input <= 4) {
		// hide UI elements when finished inputting
		this->function_entry.draw(this->_data->window);
		this->_data->window.draw(prompt);
	}

	if (finished_integrating) {
		for (Particle& p : particles) {
			p.draw(this->_data->window);
		}

		this->_data->window.draw(simulation_time);
```

```
			if (!created_from_field) {
				this->save_button.draw(this->_data->window);
				if (this->typing_name) {
					this->naming.draw(this->_data->window);
				}
			}
		}

		this->_data->window.display();
	}
```

Program.hpp

```
#pragma once

#include "StateMachine.hpp"
#include "AssetManager.hpp"
#include "InputManager.hpp"

struct ProgramData {
	StateMachine state_machine;
	AssetManager asset_manager;
	InputManager input_manager;

	sf::RenderWindow window;
};

typedef std::shared_ptr<ProgramData> ProgramDataRef;

class Program {
public:
	Program(int width, int height, std::string title);
private:
	const float dt = 1.0f / 60.0f; // frame rate = 60 fps
	sf::Clock _clock;
	ProgramDataRef _data = std::make_shared<ProgramData>();

	void run();
};
```

Program.cpp

```
#include "Program.hpp"
#include "Menu.hpp"

Program::Program(int width, int height, std::string title) {
	_data->window.create(sf::VideoMode(width, height), title, sf::Style::Close |
sf::Style::Titlebar);
	_data->state_machine.push_state(state_ref(new Menu(this->_data)));

	this->run();
}

void Program::run() {
	float new_time;
	float frame_time;
	float interpolation;

	float current_time = this->_clock.getElapsedTime().asSeconds();
	float acc = 0.0f;

	while (this->_data->window.isOpen()) {
		this->_data->state_machine.process();
		new_time = this->_clock.getElapsedTime().asSeconds();

		frame_time = new_time - current_time;
		current_time = new_time;
		acc += frame_time;

		while (acc >= dt) {
			this->_data->state_machine.get_active_state()->handle_input();
			this->_data->state_machine.get_active_state()->update(dt);

			acc -= dt;
		}

		interpolation = acc / dt;
		this->_data->state_machine.get_active_state()->draw(interpolation);
	}
}
```

RK4.hpp

```
#pragma once

#include "ShuntingYard.hpp"
#include <map>
#include <SFML/Graphics.hpp>

class Integrator {
public:
	Integrator(int steps, float end_time, ShuntingYard::Node* x_component,
ShuntingYard::Node* y_component, sf::Vector2f initial);
	void integrate();
	sf::Vector2f eval_vector_function(sf::Vector2f pos);
	sf::Vector2f get_position(float time);
private:
	std::map<float, sf::Vector2f> solution; // the solution (position at given
time)
	ShuntingYard::Node* x_component;	// x component of vector function
	ShuntingYard::Node* y_component;	// y component of vector function
	int steps;				// number of integration steps to be
taken
	float end_time;				  // time when simulation ends
};
```

RK4.cpp

```
#include "RK4.hpp"
#include <iostream>

Integrator::Integrator(int steps, float end_time,
	ShuntingYard::Node* x_component,
	ShuntingYard::Node* y_component,
	sf::Vector2f initial) {
	this->steps = steps;
	this->end_time = end_time;
	this->x_component = x_component;
	this->y_component = y_component;
	this->solution[0.0f] = initial;
}

sf::Vector2f Integrator::eval_vector_function(sf::Vector2f pos) {
	return { static_cast<float>(ShuntingYard::eval(this->x_component, pos.x,
pos.y)),
			      static_cast<float>(ShuntingYard::eval(this->y_component, pos.x,
pos.y)) };
}

void Integrator::integrate() {
	double h = end_time / steps;

	for (int step = 0; step < steps; step++) {
		sf::Vector2f point_1 = this->eval_vector_function(solution[step * h]);
		sf::Vector2f point_2 = this->eval_vector_function(solution[step * h] +
((float)h/2)*point_1);
		sf::Vector2f point_3 = this->eval_vector_function(solution[step * h] +
((float)h/2)*point_2);
		sf::Vector2f point_4 = this->eval_vector_function(solution[step * h] +
(float)h*point_3);
		solution[h * (step + 1.f)] = solution[h * step] + ((float)h / 6) *
(point_1 + 2.f * point_2 + 2.f * point_3 + point_4);
	}
}
```

```
sf::Vector2f Integrator::get_position(float time) {

	float h = end_time / steps;
	float corrected_time = h * floor(time / h); // if h=0.1 then the pos for t=1.73
wont exist, round to 1.7

	sf::Vector2f next_point = solution[h * (floor(time / h) + 1)];
	sf::Vector2f difference = next_point - solution[corrected_time];

	//std::cout << solution[corrected_time].x << ", " << solution[corrected_time].y
<< std::endl;

	//as a ratio between 0 and 1
	float fraction_passed = ((time - h * floor(time / h)) / h);  // this is "how
far" the particle has travelled along the path between the two points


	return solution[corrected_time] + fraction_passed * difference;
}
```

ShuntingYard.hpp

```
#ifndef SHUNTING_YARD
#define SHUNTING_YARD

#include <vector>
#include <string>
#include <map>
#include <stack>
#include <cmath>

namespace ShuntingYard {
	/*
		Typedefs
	*/

	// RPN list
	typedef std::vector<std::string> RPN;

	// callback to unary function (1 argument)
	typedef double(*UnaryFuncEval)(double x);

	// callback to binary function (2 arguments)
	typedef double(*BinaryFuncEval)(double x, double y);

	// types
	enum class TokenTypes {
		OPERATOR,
		CONSTANT,
		FUNCTION,
		LPAREN,
		RPAREN,
		ELSE
	};

	/*
		Utility function callbacks
	*/

	// determine if vector contains values
	template<typename T>
	bool contains(std::vector<T> v, T x);
	// obtain key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m);

	// obtain combined key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m1,
std::map<std::string, T> m2);

	// determine if character is number
	bool isNumber(char c, bool acceptDecimal = true, bool acceptNegative = true);

	// determine if entire string is number
	bool isNumber(const char* str);

	// determine if string only contains numerical characters
	bool containsNumbers(const char* str);

	// get numerical value of string
	double getNumericalVal(const char* str, float x, float y);

	// determine if string matches a function
	bool isFunction(std::string str);

	// determine if function is left associative
	bool isLeftAssociative(std::string str);

	// get function precedence
	short getPrecedence(std::string str);

	// find element from list in the equation starting at index i
	std::string findElement(int i, const char* eqn, std::vector<std::string> list);

	/*
		Function class definition
	*/
	class Func {
	public:
		// default constructor
		Func();

			// constructor for unary functions
		Func(UnaryFuncEval eval, TokenTypes type = TokenTypes::FUNCTION, short
prec = 0, bool left = true);

		// constructor for binary functions
		Func(BinaryFuncEval eval, TokenTypes type = TokenTypes::FUNCTION, short
prec = 0, bool left = true);

		double eval(double x, double y = 0);

		UnaryFuncEval u_eval;       // unary function evaluation callback
		BinaryFuncEval b_eval;      // binary function evaluation callback

		TokenTypes type;            // type of function (ie function or operator)
		short prec;                        // precedence
		bool left;                         // is left associative
		bool unary;                        // is a unary function

	private:
		Func(TokenTypes type, short prec, bool left, bool unary);
};

/*
	Reference
*/

// unary functions


/*
	Node class definitions
*/

// base node class
class Node {
public:
	Node(std::string name, bool isFunc);

	double eval(double x = 0, double y = 0);

	std::string name;
	bool isFunc;

	Node* right;
	Node* left;
};

// function node class
class FuncNode : public Node {
public:
	FuncNode(std::string name);

	// set type of function and then assign callback
	void setUnary(bool isUnary);

	// evaluate
	double eval(double x, double y = 0);

	bool isUnary;
	Func func;
};

// number node class
class NumNode : public Node {
public:
	NumNode(std::string name);

	// return numerical value
	double eval(double x = 0, double y = 0);
};

/*
	Main functions
*/

// parse infix notation into reverse polish notation (Shunting Yard)
RPN reversePolishNotation(const char* eqn);

// parse RPN to tree
Node* parse(RPN rpn);

// evaluate tree
```

```
	double eval(Node* tree, float x, float y);

	/*
		Utility function definitions
	*/

	// determine if vector contains values
	template<typename T>
	bool contains(std::vector<T> v, T x);

	// obtain key list
	template<typename T>
	inline std::vector<std::string> keys(std::map<std::string, T> m);

	// obtain combined key list
	template<typename T>
	std::vector<std::string> keys(std::map<std::string, T> m1,
std::map<std::string, T> m2);

	// determine if character is number
	bool isNumber(char c, bool acceptDecimal, bool acceptNegative);

	// determine if entire string is number
	bool isNumber(const char* str);

	// determine if string only contains numerical characters
	bool containsNumbers(const char* str);

	// get numerical value of string
	double getNumericalVal(const char* str, float x, float y);

	// determine if string matches a function
	bool isFunction(std::string str);

	// determine if function is left associative
	bool isLeftAssociative(std::string str);

	// get function precedence
	short getPrecedence(std::string str);

	// find element from list in the equation starting at index i
	std::string findElement(int i, const char* eqn, std::vector<std::string> list);
}

#endif
```

ShuntingYard.cpp

```
#include "ShuntingYard.hpp"
#include <iostream>

namespace ShuntingYard {
	std::map<std::string, Func> unary_functions = {
		{ "sin", Func(std::sin) }
	};
	// binary functions
	std::map<std::string, Func> binary_functions = {
		{ "+", Func([](double x, double y) -> double { return x + y; },
TokenTypes::OPERATOR, 2) },
		{ "-", Func([](double x, double y) -> double { return x - y; },
TokenTypes::OPERATOR, 2) },
		{ "*", Func([](double x, double y) -> double { return x * y; },
TokenTypes::OPERATOR, 3) },
		{ "/", Func([](double x, double y) -> double { return x / y; },
	TokenTypes::OPERATOR, 3) },
		{ "^", Func(std::pow, TokenTypes::OPERATOR, 4, false) }
	};

	// function names
	std::vector<std::string> functionNames = keys<Func>(unary_functions,
	binary_functions);

	// constants
	std::map<std::string, double> constants = {
		{ "pi", std::atan(1) * 4 },
		{ "e", std::exp(1) }
	};

	// constant names
	std::vector<std::string> constantNames = keys<double>(constants);

	// variables
	std::map<std::string, double> variables;

	// operators
	std::vector<char> operators = { '+', '-', '/', '*', '^' };
	// left brackets
	std::vector<char> leftBrackets = { '(', '{', '[' };
	// right brackets
	std::vector<char> rightBrackets = { ')', '}', ']' };

	Func::Func()
		: type(TokenTypes::OPERATOR), prec(0), left(true), unary(true),
	u_eval(nullptr), b_eval(nullptr) {}

	Func::Func(UnaryFuncEval eval, TokenTypes type, short prec, bool left)
		: Func(type, prec, left, true) {
		u_eval = eval;
	}

	Func::Func(BinaryFuncEval eval, TokenTypes type, short prec, bool left)
		: Func(type, prec, left, false) {
		b_eval = eval;
	}

	double Func::eval(double x, double y) {
		return this->unary ? u_eval(x) : b_eval(x, y);
	}

	Func::Func(TokenTypes type, short prec, bool left, bool unary)
		: type(type), prec(prec), left(left), unary(unary), u_eval(nullptr),
	b_eval(b_eval) {}

	Node::Node(std::string name, bool isFunc)
		: name(name), isFunc(isFunc) {}

	FuncNode::FuncNode(std::string name)
		: Node(name, true) {}

	void FuncNode::setUnary(bool isUnary) {
		this->isUnary = isUnary;

		this->func = isUnary ? unary_functions[name] : binary_functions[name];
	}
```

```
        double FuncNode::eval(double x, double y) {
                return this->func.eval(x, y);
        }

        NumNode::NumNode(std::string name)
                : Node(name, false) {}

        double NumNode::eval(double x, double y) {
                return getNumericalVal(name.c_str(), x, y);
        }

        RPN reversePolishNotation(const char* eqn) {
                std::vector<std::string> queue;
                std::stack<std::string> stack;

                std::string obj = "";
                TokenTypes type = TokenTypes::ELSE;
                TokenTypes prevType = TokenTypes::ELSE; // negative sign detection

                bool acceptDecimal = true;
                bool acceptNegative = true;

                // token reading and detection
                for (int i = 0, eqLen = (int)strlen(eqn); i < eqLen; i++) {
                        char t = eqn[i];

                        // skip spaces and commas
                        if (t == ' ' || t == ',') {
                                //prevType = TokenTypes::ELSE;
                                continue;
                        }

                        // classify token
                        if (isNumber(t)) {
                                type = TokenTypes::CONSTANT;
                                if (t == '.') {
                                        acceptDecimal = false;
                                }
                                else if (t == '-') {
                                        acceptNegative = false;
                                }

                                int startI = i;
                                if (i < eqLen - 1) {
                                        while (isNumber(eqn[i + 1], acceptDecimal,
acceptNegative)) {
                                                i++;
                                                if (i >= eqLen - 1) {
                                                        break;
                                                }
                                        }
                                }
                                obj = std::string(eqn).substr(startI, i - startI + 1);

                                // subtraction sign detection
                                if (obj == "-") {
                                        type = TokenTypes::OPERATOR;
                                }
                        }
                        else {
                                obj = findElement(i, eqn, functionNames);
                                if (obj != "") {
```

```
                                // found valid object
                                type = contains<char>(operators, obj[0]) ?
TokenTypes::OPERATOR : TokenTypes::FUNCTION;
                        }
                        else {
                                obj = findElement(i, eqn, constantNames);
                                if (obj != "") {
                                        // found valid object
                                        type = TokenTypes::CONSTANT;
                                }
                                else {
                                        obj = findElement(i, eqn, {"x", "y"});
                                        if (obj != "") {
                                                type = TokenTypes::CONSTANT;
                                        }
                                        else if (contains<char>(leftBrackets, t)) {
                                                type = TokenTypes::LPAREN;
                                                obj = "(";
                                        }
                                        else if (contains<char>(rightBrackets, t)) {
                                                type = TokenTypes::RPAREN;
                                                obj = ")";
                                        }
                                        else {
                                                type = TokenTypes::ELSE;
                                        }
                                }
                        }
                        i += obj.size() - 1;
                }

                // do something with the token
                const char* last_stack = (stack.size() > 0) ? stack.top().c_str()
: "";

                switch (type) {
                case TokenTypes::CONSTANT:
                        queue.push_back(obj);
                        break;
                case TokenTypes::FUNCTION:
                        stack.push(obj);
                        break;
                case TokenTypes::OPERATOR:
                        if (stack.size() != 0) {
                                while (
                                        (
                        (contains<std::string>(functionNames, last_stack) &&
                                        !contains<char>(operators, last_stack[0])) ||
                                getPrecedence(last_stack) > getPrecedence(obj) ||
                        ((getPrecedence(last_stack) == getPrecedence(obj)) &&
                isLeftAssociative(last_stack))) && !contains<char>(leftBrackets,
last_stack[0])
                                        ) {
                                        // pop from the stack to the queue
                                        queue.push_back(stack.top());
                                        stack.pop();
                                        if (stack.size() == 0) {
                                                break;
                                        }
                                        last_stack = stack.top().c_str();
                                }
                        }
```

```
	stack.push(obj);
					break;
				case TokenTypes::LPAREN:
					stack.push(obj);
					break;
				case TokenTypes::RPAREN:
					while (last_stack[0] != '(') {
						// pop from the stack to the queue
						queue.push_back(stack.top());
						stack.pop();
						last_stack = stack.top().c_str();
					}
					stack.pop();
					break;
				default:
					return queue;
				}

				// reset type
				prevType = type;
			}

			while (stack.size() > 0) {
				// pop from the stack to the queue
				queue.push_back(stack.top());
				stack.pop();
			}

			return queue;
		}

		// parse RPN to tree
		Node* parse(RPN rpn) {
			std::stack<Node*> stack;

			for (std::string item : rpn) {
				if (isNumber(item.c_str())) {
					// push number node
					stack.push(new NumNode(item));
				}
				else {
					// function
					FuncNode* f = new FuncNode(item);
					if (contains<std::string>(keys(binary_functions), item)) {
						f->setUnary(false);
						// set children of node

						// right child is second argument
						f->right = stack.top();
						stack.pop();

						// left child is first argument
						f->left = stack.top();
						stack.pop();
					}
					else if (contains<std::string>(keys(unary_functions),
	item)) {
						f->setUnary(true);
						// set child of node
						f->left = stack.top();
						stack.pop();
					}
```

```
                        stack.push(f);
                    }
                }

                if (stack.size() == 0) {
                    return nullptr;
                }

                return stack.top();
            }

            // evaluate tree
            double eval(Node* tree, float x, float y) {
                if (tree->isFunc) {
                    FuncNode* ftree = (FuncNode*)tree;
                    if (ftree->isUnary) {
                        // evaluate child recursively and then evaluate with
        return value
                        return ftree->eval(eval(tree->left, x, y));
                    }
                    else {
                        // evaluate each child recursively and then evaluate with
        return value
                        return ftree->eval(eval(tree->left, x, y), eval(tree-
        >right, x, y));
                    }
                }
                else {
                    // number node
                    return ((NumNode*)tree)->eval(x, y);
                }
            }

            /*
                Utility function definitions
            */

            // determine if vector contains values
            template<typename T>
            bool contains(std::vector<T> v, T x) {
                return std::find(v.begin(), v.end(), x) != v.end();
            }

            // obtain key list
            template<typename T>
            std::vector<std::string> keys(std::map<std::string, T> m) {
                std::vector<std::string> ret;

                // push each key from each pair
                for (auto const& element : m) {
                    ret.push_back(element.first);
                }

                return ret;
            }

            // obtain combined key list
            template<typename T>
            std::vector<std::string> keys(std::map<std::string, T> m1,
        std::map<std::string, T> m2) {
                // get keys from each map
                std::vector<std::string> keySet1 = keys<T>(m1);
```

```
		std::vector<std::string> keySet2 = keys<T>(m2);

		// insert the second list into first
		keySet1.insert(keySet1.end(), keySet2.begin(), keySet2.end());

		// return result
		return keySet1;
	}

	// determine if character is number
	bool isNumber(char c, bool acceptDecimal, bool acceptNegative) {
		// digits
		if (c >= '0' && c <= '9') {
			return true;
		}
		// decimal point
		else if (acceptDecimal && c == '.') {
			return true;
		}
		// negative sign
		else if (acceptNegative && c == '-') {
			return true;
		}

		return false;
	}

	// determine if entire string is number
	bool isNumber(const char* str) {
		// it's a constant, variable, or a numerical string
		return contains<std::string>(constantNames, str) ||
			contains<std::string>({"x", "y"}, str) ||
			containsNumbers(str);
	}

	// determine if string only contains numerical characters
	bool containsNumbers(const char* str) {
		// cannot be a single decimal point or negative sign
		if (std::strcmp(str, ".") == 0 || std::strcmp(str, "-") == 0) {
			return false;
		}

		std::string obj = std::string(str);

		// try to prove wrong
		bool acceptDecimal = true;
		if (isNumber(obj[0], true, true)) {
			// check first character for negative sign
			if (obj[0] == '.') {
				// cannot be any more decimal points
				acceptDecimal = false;
			}
		}
		else {
			return false;
		}

		for (unsigned int i = 1, len = obj.size(); i < len; i++) {
			// do not accept anymore negative signs
			if (!isNumber(obj[i], acceptDecimal, false)) {
				return false;
			}
```

```
                        if (obj[i] == '.') {
                              // cannot be any more decimal points
                              acceptDecimal = false;
                        }
                  }

                  return true;
            }

            // get numerical value of string
            double getNumericalVal(const char* str, float x = 0, float y = 0) {
                  if (contains<std::string>(constantNames, str)) {
                        // is a constant
                        return constants[str];
                  }
                  else if (std::string(str) == "x") {
                        return x;
                  }
                  else if (std::string(str) == "y") { return y; }
                  else {
                        // is a number
                        return std::atof(str);
                  }
            }

            // determine if string matches a function
            bool isFunction(std::string str) {
                  return contains<std::string>(functionNames, str);
            }

            // determine if function is left associative
            bool isLeftAssociative(std::string str) {
                  return binary_functions[str].left;
            }

            // get function precedence
            short getPrecedence(std::string str) {
                  if (contains<std::string>(keys(binary_functions), str)) {
                        return binary_functions[str].prec;
                  }

                  // only care about operators, which are binary functions, so otherwise
      we can return 0
                  return 0;
            }

            // find element from list in the equation starting at index i
            std::string findElement(int i, const char* eqn, std::vector<std::string> list)
      {
                  for (std::string item : list) {
                        int n = (int)item.size();
                        if (std::string(eqn).substr(i, n) == item) {
                              return item;
                        }
                  }

                  return "";
            }
      }
```

State.hpp

```
#pragma once

class State {
public:
	virtual void init() = 0;
	virtual void handle_input() = 0;
	virtual void update(float dt) = 0;
	virtual void draw(float dt) = 0;

	virtual void pause() {}
	virtual void resume() {}
};
```

StateMachine.hpp

```
#pragma once

#include <memory>
#include <stack>
#include "State.hpp"

typedef std::unique_ptr<State> state_ref;

class StateMachine {
public:
	StateMachine() {}
	~StateMachine() {}
	void push_state(state_ref new_state, bool is_replacing = true);
	void pop_state();
	void process();
	state_ref& get_active_state();

private:
	std::stack<state_ref> _states;
	state_ref _new_state;
	bool _is_removing;
	bool _is_adding;
	bool _is_replacing;
};
```

StateMachine.cpp

```cpp
#include "StateMachine.hpp"

void StateMachine::push_state(state_ref new_state, bool is_replacing) {
	this->_is_adding = true;
	this->_is_replacing = is_replacing;
	this->_new_state = std::move(new_state);
}

void StateMachine::pop_state() {
	this->_is_removing = true;
}

void StateMachine::process() {
	if (this->_is_removing && !this->_states.empty()) {
		this->_states.pop();

		if(!this->_states.empty()) {
			this->_states.top()->resume();
		}

		this->_is_removing = false;
	}

	if (this->_is_adding) {
		if (!this->_states.empty()) {
			if (this->_is_replacing) {
				this->_states.pop();
			} else {
				this->_states.top()->pause();
			}
		}

		this->_states.push(std::move(this->_new_state));
		this->_states.top()->init();
		this->_is_adding = false;
	}
}

state_ref& StateMachine::get_active_state() {
	return this->_states.top();
}
```

Textbox.hpp

```
#pragma once

#include <sstream>
#include <SFML/Graphics.hpp>

#define BACKSPACE 8 // ascii code for backspace
#define ENTER 13

class Textbox {
public:
	Textbox(int size, sf::Color color, bool selected);
	void set_font(sf::Font& font);
	void set_position(sf::Vector2f pos);
	void set_limit(bool limited);
	void set_limit(int limit);
	void set_selected(bool selected);
	void set_text(std::string str);
	std::string get_text();
	void draw(sf::RenderWindow& window);
	void on_type(sf::Event event);
private:
	sf::Text textbox;
	std::ostringstream text;
	bool is_selected = false;
	bool has_limit = false;
	int limit;

	void input_logic(int char_typed);
	void delete_last_char();
};
```

Textbox.cpp

```
#include "Textbox.hpp"

Textbox::Textbox(int size, sf::Color color, bool selected) {
	this->textbox.setCharacterSize(size);
	this->textbox.setFillColor(color);
	this->is_selected = selected;

	if (selected) {
		textbox.setString("_");
	} else {
		textbox.setString("");
	}
}

void Textbox::input_logic(int char_typed) {
	if (char_typed == BACKSPACE) {
		if (text.str().length() > 0) {
			this->delete_last_char();
		}
	} else if (char_typed != ENTER) {
		text << static_cast<char>(char_typed);
	}

	textbox.setString(text.str() + "_");
}
```

```
void Textbox::delete_last_char() {
	std::string temp = text.str();
	std::string new_text = "";

	for (int i = 0; i < temp.length() - 1; i++) {
		//add all characters except for last one
		new_text += temp[i];
	}

	text.str(""); // clear current text
	text << new_text; // append to text
}

void Textbox::set_font(sf::Font& font) {
	this->textbox.setFont(font);
}

void Textbox::set_position(sf::Vector2f pos) {
	this->textbox.setPosition(pos);
}

void Textbox::set_limit(bool limited) {
	this->has_limit = limited;
}

void Textbox::set_limit(int limit) {
	this->has_limit = true;
	this->limit = limit;
}

void Textbox::set_selected(bool selected) {
	this->is_selected = selected;
	textbox.setString(text.str() + "_");

	if (!selected) {
		// if its not selected we dont want the _ at the end
		std::string temp = text.str();
		std::string new_text = "";

		for (int i = 0; i < temp.length() - 1; i++) {
			//add all characters except for last one
			new_text += temp[i];
		}
		textbox.setString(new_text);
	}
}

void Textbox::set_text(std::string str) {
	text.str(""); // clear texsdt stream to nothing
	text << str; // append full string to new stream
	textbox.setString(str+"_"); // set displayed text to the string
}

std::string Textbox::get_text() {
	return this->text.str();
}

void Textbox::draw(sf::RenderWindow& window) {
	window.draw(textbox);
}

void Textbox::on_type(sf::Event event) {
        if (this->is_selected) {
              int char_typed = event.text.unicode;
              if (char_typed < 128) {
                    // allow ascii only
                    if (this->has_limit) {
                          if (text.str().length() <= limit) {
                                this->input_logic(char_typed);
                          } else if(text.str().length() > limit && char_typed ==
  BACKSPACE) {
                                //allow them to delete chracters when over the
  limit
                                this->delete_last_char();
                          }
                    } else {
                          this->input_logic(char_typed);
                    }
              }
        }
  }
```