International Baccalaureate

IB Computer Science SL

Chemistry Equilibrium Simulator Web-Based App

Internal Assessment

Candidate Code: lny464

Word Count: 1951

Number of Pages: 48

April 10, 2025

(Word count for Scenario, Rationale, Success Criteria: 628 words)

Scenario

My client is my AP Chemistry teacher at Skyline High School, Mr. Gunderson. An interview with my client on December 10, 2024, provided insights into how many students struggle with the chapter on chemical equilibrium, particularly with visualizing how reactant and product concentrations change over time. Mr. Gunderson noted that while textbooks provide static graphs and diagrams, students often find it difficult to conceptualize the dynamic nature of reversible reactions and the concept of equilibrium constants (K). He expressed a need for an interactive learning tool that could simulate these chemical systems in real time and allow students to manipulate variables such as initial concentrations, temperature, or reaction type. He also wanted an easily accessible tool that could calculate crucial quantities in the equilibrium chapter, as well as provide students with basic principles. This would help bridge the gap between theoretical knowledge and practical understanding. Based on this, I proposed developing a web-based chemical equilibrium simulator that would provide real-time feedback, data visualization, and a user-friendly interface to assist both teaching and learning in the classroom

Rationale of the Proposed Solution

A web-based application is the most appropriate solution for the environment at Skyline High School, where students are issued school-managed Chromebooks. These devices are subject to strict software restrictions, making it impossible to install standalone applications or run unapproved software. Since all Chromebooks have access to the internet and modern browsers, a web application can be universally accessed by all students without requiring any changes to their devices. Hosting the project online using GitHub Pages ensures easy deployment and continuous availability, regardless of whether students are at school or working from home.

In addition to being accessible, a web-based solution eliminates the common barriers associated with traditional software installations, such as compatibility issues, storage limitations, and user permissions. By simply visiting a URL, students can interact with the simulator in real time. This ease of access not only saves time for both students and teachers but also ensures that no instructional time is lost dealing with technical issues. The choice to use standard web technologies—HTML, CSS, and JavaScript—supports a lightweight, fast-loading application that works reliably across different devices and screen sizes.

Finally, the interactive nature of a web-based simulation enhances the educational experience by making abstract chemical equilibrium concepts more tangible. The client, Mr. Gunderson, emphasized the difficulty students face when trying to visualize how concentration levels shift during reversible reactions. This application addresses that challenge by allowing users to input values, observe changes graphically, and interact with the system in real time. The solution is not only technically feasible within the constraints of the school system but also pedagogically effective, as it encourages exploration and active learning.

Success Criteria

1. The user must be able to input or select initial concentrations for at least one reversible reaction.
2. The application must simulate the reaction reaching equilibrium and visually indicate when equilibrium is achieved.

3. The system must dynamically display changes in concentration over time using a graphical representation (e.g., chart or bar graph).
4. The equilibrium constant (K) must be automatically calculated and displayed based on the concentrations entered.
5. The simulation must allow the user to reset and re-run the reaction with different input values.
6. The user interface must be clear, labeled, and understandable by high school students without requiring external instructions.
7. The application must load and run directly in a browser without requiring any installations, working fully through GitHub Pages.
8. The app must be responsive and function properly on school-managed Chromebooks using the latest version of Google Chrome.
9. Invalid input values (such as negative concentrations or empty fields) must be handled gracefully with appropriate error messages.
10. The client, Mr. Gunderson, must confirm that the app aligns with his teaching goals and can be used effectively in a classroom setting.

Record of Tasks

Task #	Planned Action	Planned Outcome	Time Estimated (hrs)	Target Completion Date	Criteri on
1	Interview client to understand the problem	Notes from meeting and client-identified needs	1	Dec 10, 2024	A
2	Write and revise problem statement and rationale	Clear problem definition aligned with client needs	1	Dec 11, 2024	A
3	Define success criteria based on functionality, constraints, and goals	List of specific and measurable success criteria	1	Dec 12, 2024	A
4	Research equilibrium theory and online visual learning tools	Notes and verified chemical behavior to simulate accurately	1.5	Dec 14, 2024	B
5	Sketch user interface wireframes	Basic UI layout (inputs, display area, buttons)	1	Dec 15, 2024	B
6	Choose programming tools and deployment method	Finalized decision: HTML/CSS/JS + GitHub Pages	0.5	Dec 17, 2024	B
7	Set up GitHub	Project environment	1.5	Dec 18, 2024	C

	repository and initial project files	with base HTML/CSS			
8	Build input form in HTML and style with CSS	Interface ready for user data entry	2	Dec 20, 2024	C
9	Write JavaScript to collect and validate user input	Inputs stored correctly with validation	2	Dec 22, 2024	C
10	Add error handling for invalid/empty values	Error messages for incorrect input	1.5	Dec 23, 2024	C
11	Add UI elements for displaying K value and units	Clear, real-time display of equilibrium constant	1	Dec 24, 2024	C
12	Test K calculation for different reaction scenarios	Ensure formula works with common and edge cases	1.5	Dec 25, 2024	D
13	Style and label all inputs and outputs clearly	Improve student usability and accessibility	1	Dec 25, 2024	C
14	Create mock reactions (e.g., A + B ⇌ C) with preset values	Allow quick demos for teacher or students	1.5	Dec 26, 2024	C
15	Create table layout to track concentration changes step-by-step	Optional numeric view for detail-oriented learners	2	Dec 26, 2024	C
16	Add adjustable simulation speed slider	Allow students to slow or speed up progression	1	Dec 27, 2024	C
17	Debug inconsistencies in K value due to rounding	Ensure values are reliable and precise	1	Dec 27, 2024	D
18	Add confirmation before resetting simulation	Prevent accidental loss of data	0.5	Dec 28, 2024	C
19	Use CSS transitions to animate concentration bars	Make visualization more intuitive	1	Dec 28, 2024	C
20	Begin drafting developer documentation for future updates	Brief guide on how to edit/extend project	1	Dec 29, 2024	D

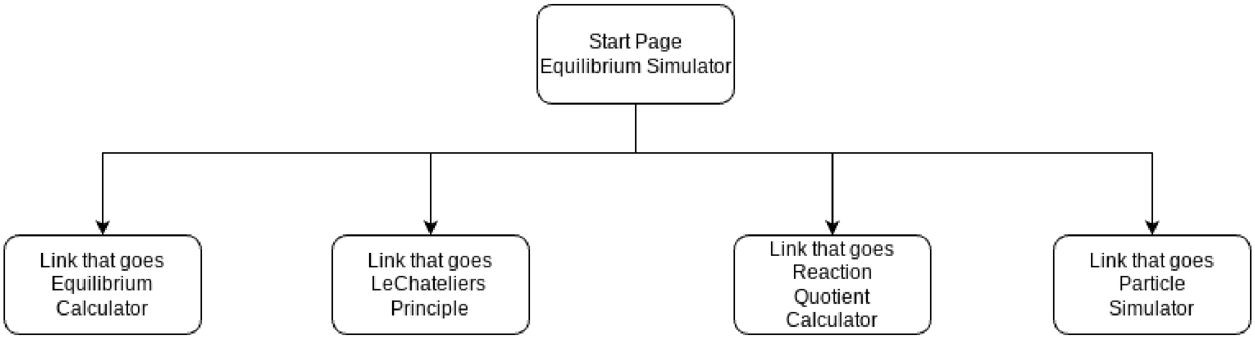
21	Code the simulation of equilibrium process	Dynamic concentration updates over time	2.5	Dec 30, 2024	C
22	Implement logic to calculate equilibrium constant (K)	Real-time display of K value based on concentrations	1.5	Dec 30, 2024	C
23	Link input/output to update dynamically	User changes cause live simulation response	2	Jan 2, 2025	C
24	Add bar/graph visualization of concentration changes	Visual representation of reaction dynamics	2.5	Jan 2, 2025	C
25	Create visual or textual indicator of equilibrium being reached	Message or UI effect triggered when K is stable	1	Jan 3, 2025	C
26	Create reset button to restart simulation	New input resets graph and values	1	Jan 3, 2025	C
27	Make layout responsive for various screen sizes	Works well on Chromebooks and desktops	1.5	Jan 4, 2025	C
28	Deploy site using GitHub Pages	Public URL accessible for testing	1	Jan 4, 2025	D
29	Test site performance and appearance on school Chromebook	Verified compatibility with school devices	1	Jan 5, 2025	D
30	Gather client feedback on usability and accuracy	Notes and suggestions from Mr. Gunderson	1	Jan 6, 2025	D
31	Update UI and functionality based on feedback	Improved version matching teacher expectations	1	Jan 7, 2025	C
32	Run additional tests with edge case inputs	Verified correct behavior with low, high, and invalid values	1.5	Jan 8, 2025	D
33	Add tooltips and short instructional text	Students can understand UI with minimal help	1	Jan 9, 2025	C
34	Final testing and	Finished version ready	1	Jan 10, 2025	C

	cleanup of visuals, functions, layout	for classroom use			
35	Take annotated screenshots of working solution	Visual evidence for report documentation	1	Jan 12, 2025	D
36	Write Planning and Design sections of the IA report	Initial sections of report completed	2	Jan 13, 2025	A, B
37	Write and explain Development section with screenshots and code	Development section fully documented	2	Jan 14, 2025	C
38	Reflect on success criteria and write Evaluation section	Evaluation of how well solution met goals	1.5	Jan 15, 2025	E
39	Review final report and edit with supervisor feedback	Polished final draft	1	Jan 16, 2025	E
40	Submit IA report, code, and documentation	Final version officially submitted	0.5	Jan 17, 2025	E

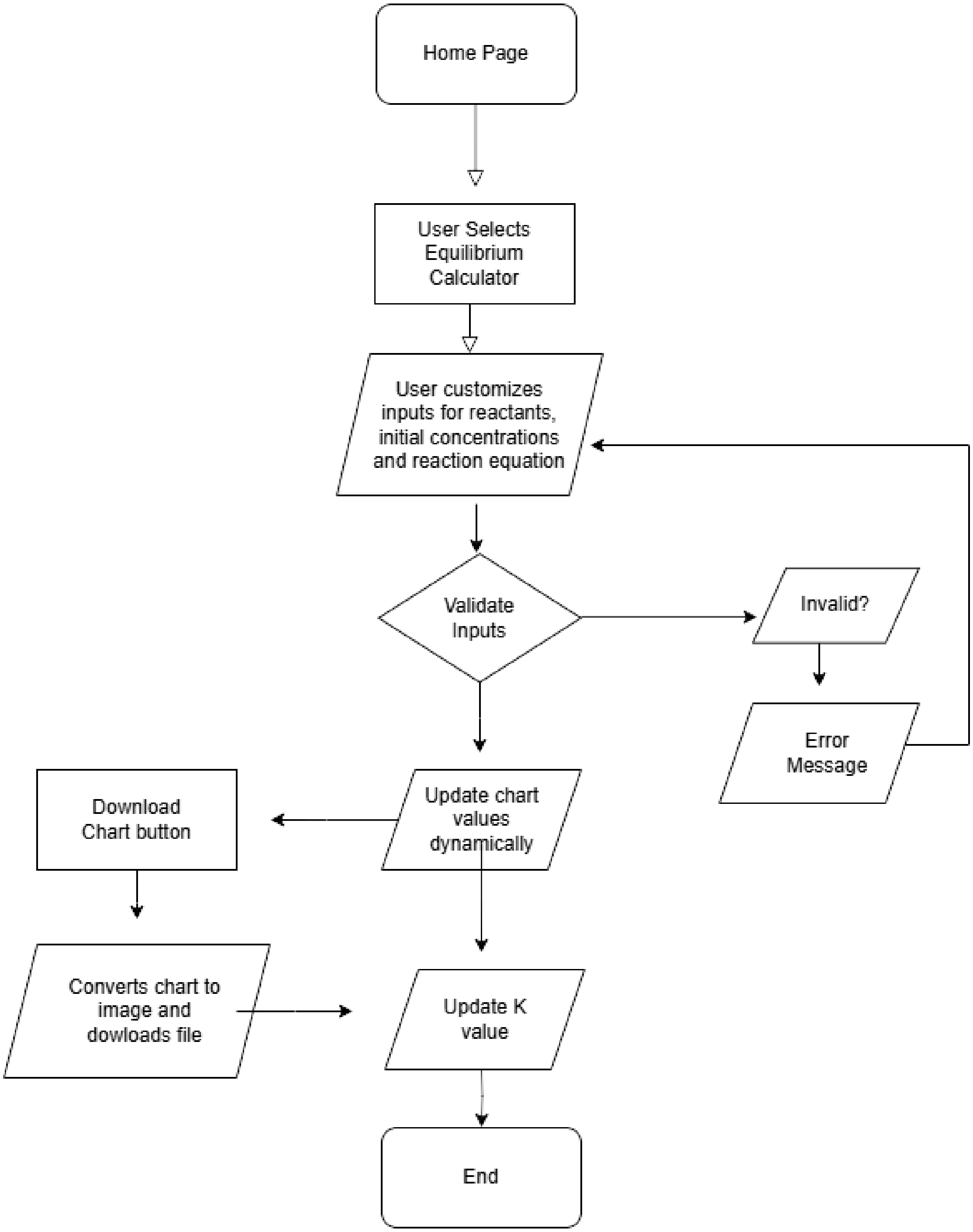
Design

Flowcharts outlining the overall process

General Process

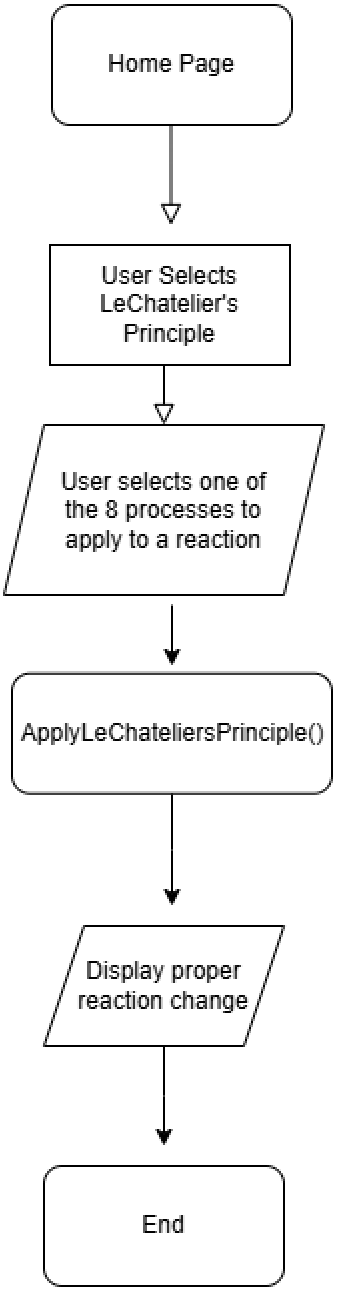

Start Page
Equilibrium Simulator
Link that goes Equilibrium Calculator
Link that goes LeChateliers Principle
Link that goes Reaction Quotient Calculator
Link that goes Particle Simulator

Equilibrium Calculator Page

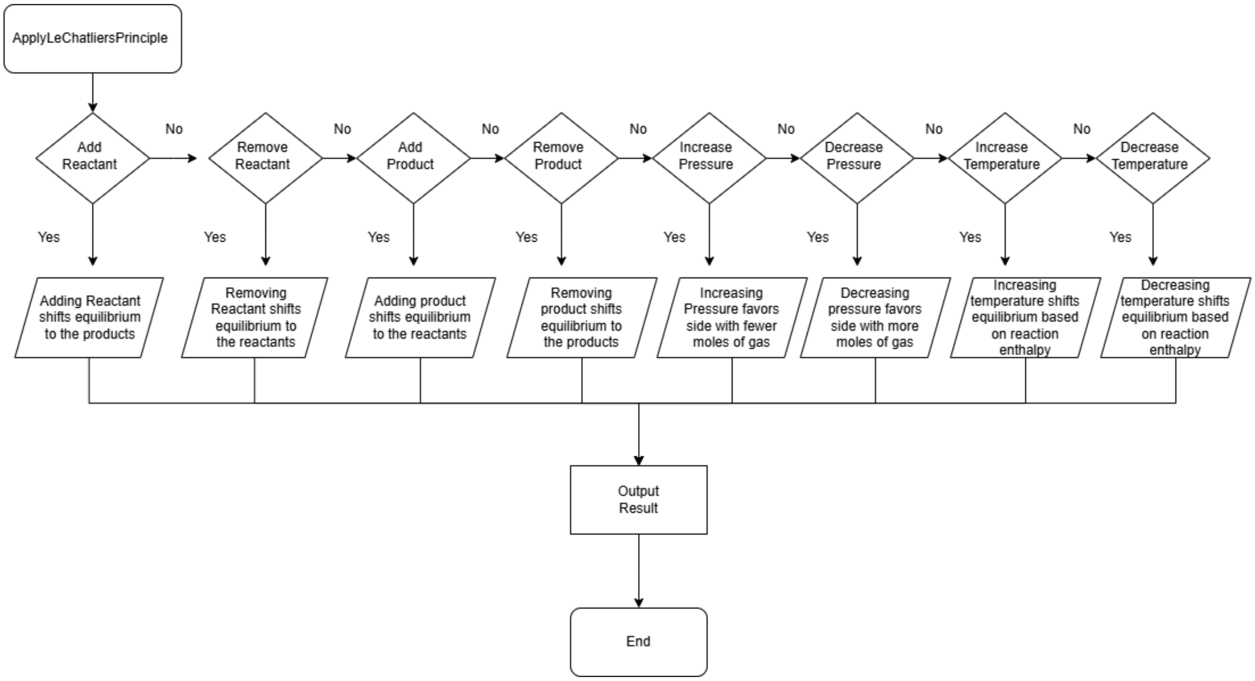


Home Page
User Selects Equilibrium Calculator
User customizes inputs for reactants, initial concentrations and reaction equation
Validate Inputs
Invalid?
Error Message
Update chart values dynamically
Download Chart button
Converts chart to image and dowloads file
Update K value
End

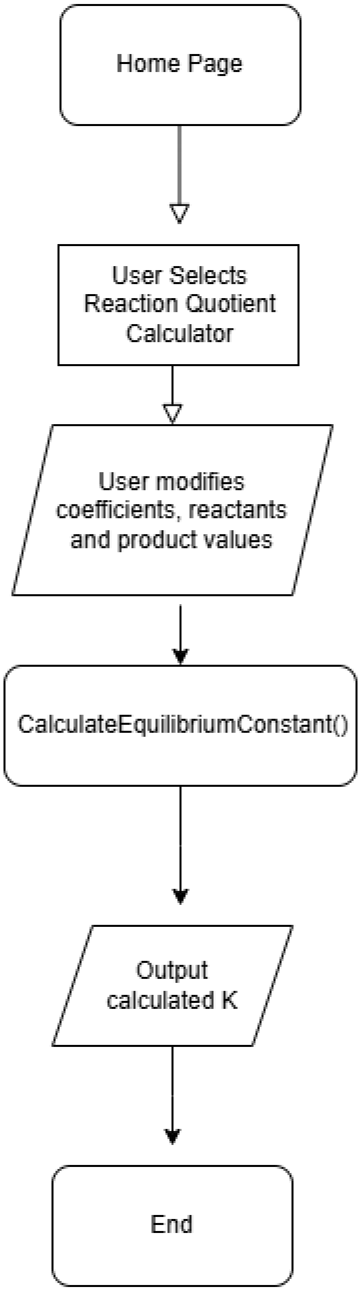
LeChatelier’s Principle page


Home Page
User Selects LeChatelier's Principle
User selects one of the 8 processes to apply to a reaction
ApplyLeChateliersPrinciple()
Display proper reaction change
End

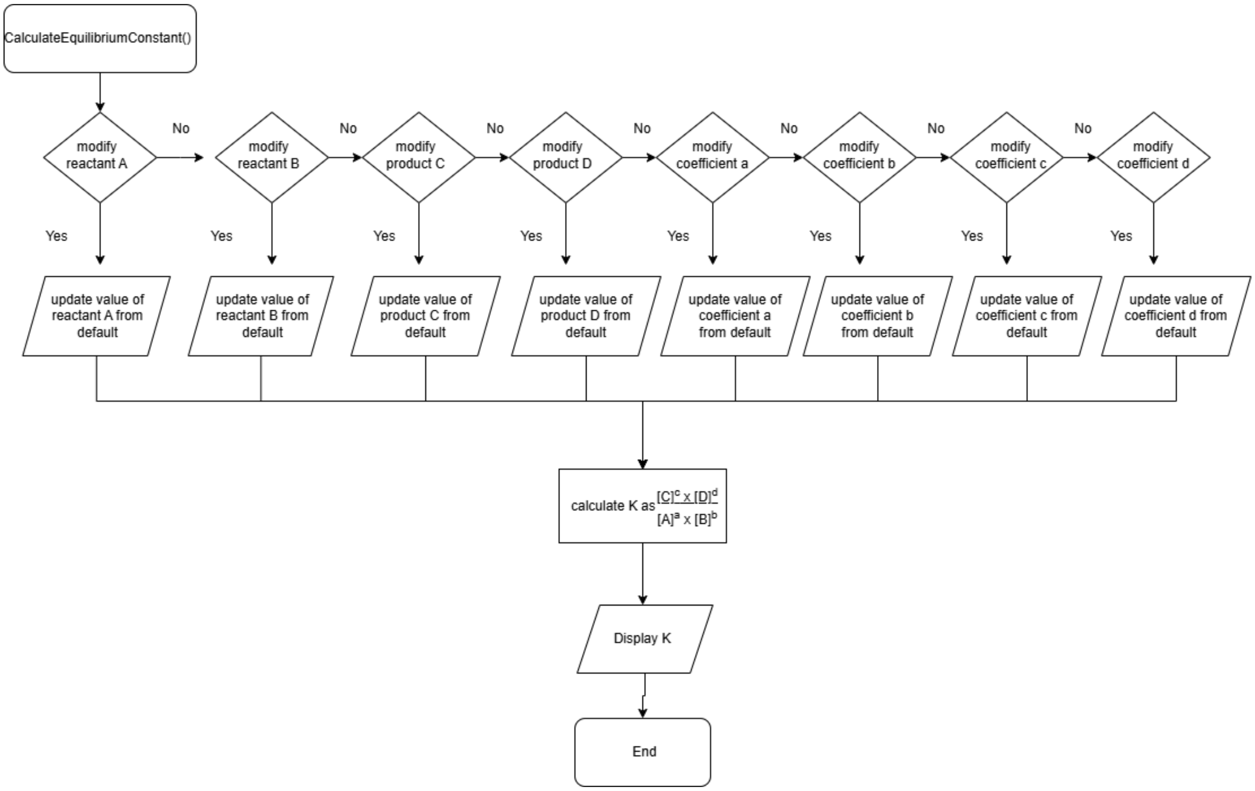
ApplyLeChateliersPrinciple()


ApplyLeChatliersPrinciple
Add Reactant
No
Remove Reactant
No
Add Product
No
Remove Product
No
Increase Pressure
No
Decrease Pressure
No
Increase Temperature
No
Decrease Temperature
Yes
Yes
Yes
Yes
Yes
Yes
Yes
Yes
Adding Reactant shifts equilibrium to the products
Removing Reactant shifts equilibrium to the reactants
Adding product shifts equilibrium to the reactants
Removing product shifts equilibrium to the products
Increasing Pressure favors side with fewer moles of gas
Decreasing pressure favors side with more moles of gas
Increasing temperature shifts equilibrium based on reaction enthalpy
Decreasing temperature shifts equilibrium based on reaction enthalpy
Output Result
End

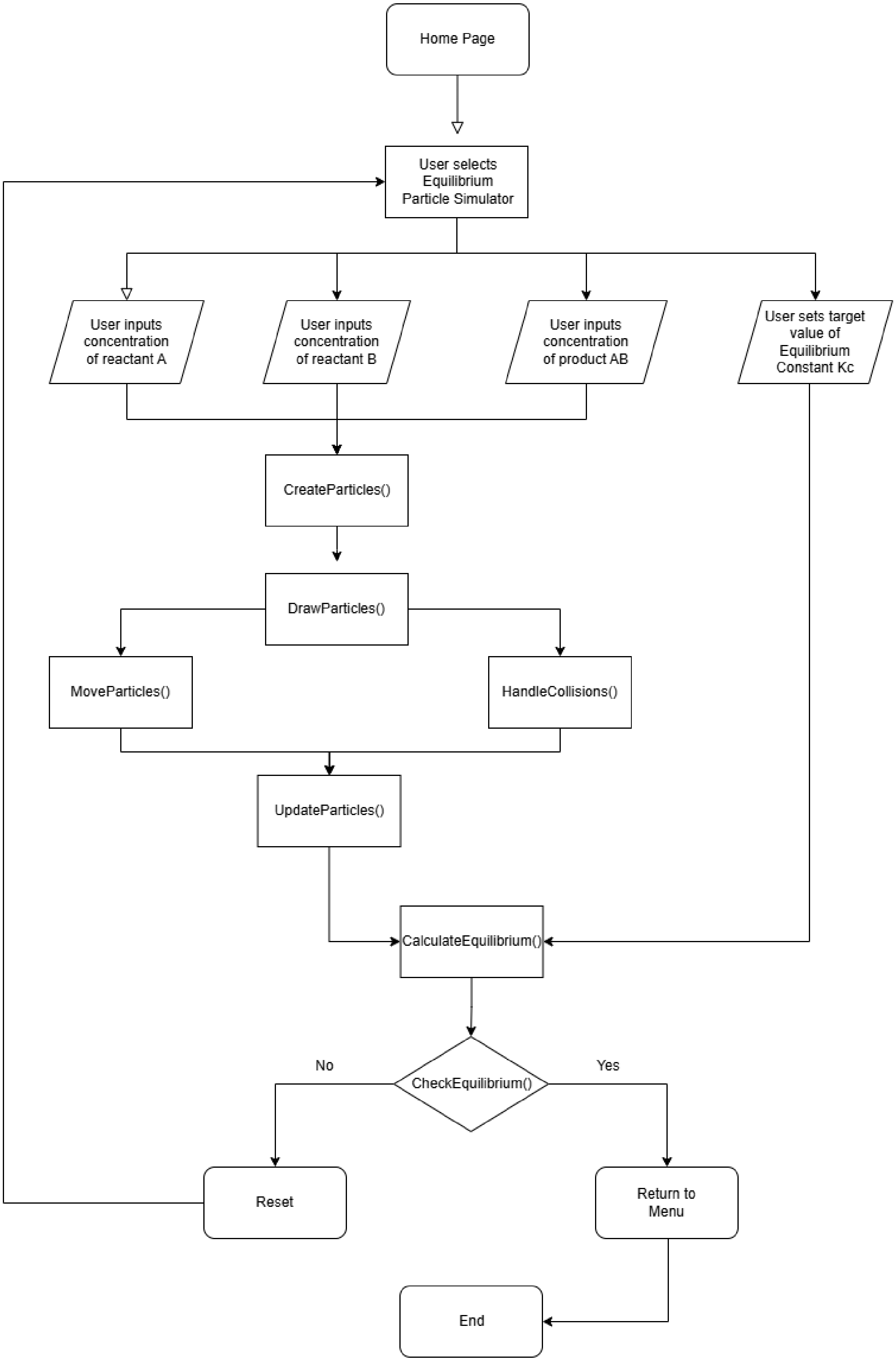
ReactionQuotientCalculator page


Home Page
User Selects Reaction Quotient Calculator
User modifies coefficients, reactants and product values
CalculateEquilibriumConstant()
Output calculated K
End

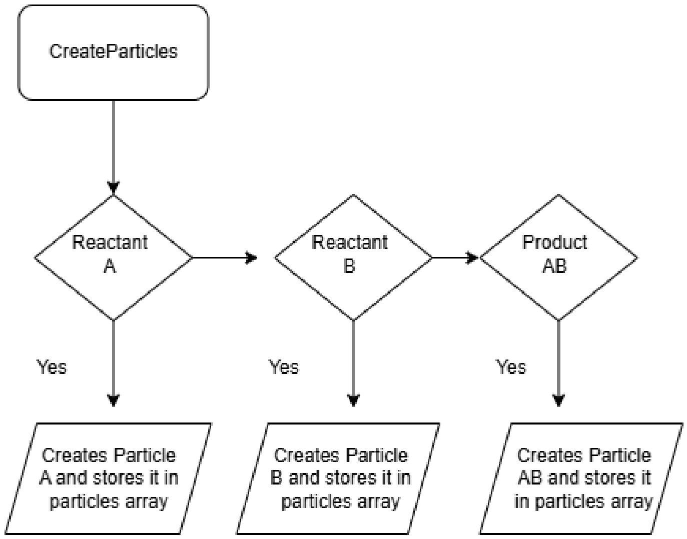
CalculateEquilibriumConstant()

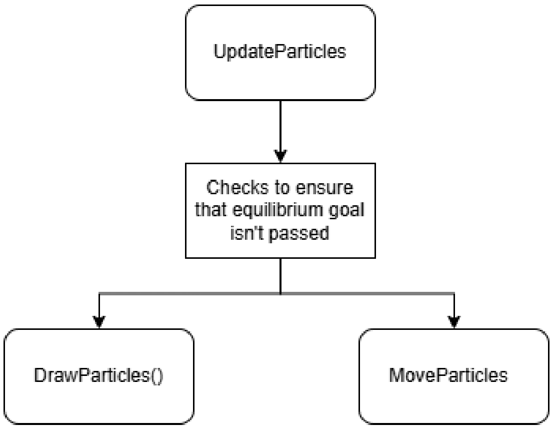

CalculateEquilibriumConstant()
modify reactant A
No
modify reactant B
No
modify product C
No
modify product D
No
modify coefficient a
No
modify coefficient b
No
modify coefficient c
No
modify coefficient d
Yes
Yes
Yes
Yes
Yes
Yes
Yes
Yes
update value of reactant A from default
update value of reactant B from default
update value of product C from default
update value of product D from default
update value of coefficient a from default
update value of coefficient b from default
update value of coefficient c from default
update value of coefficient d from default
calculate K as $\frac{[C]^c \text{ x } [D]^d}{[A]^a \text{ x } [B]^b}$
Display K
End

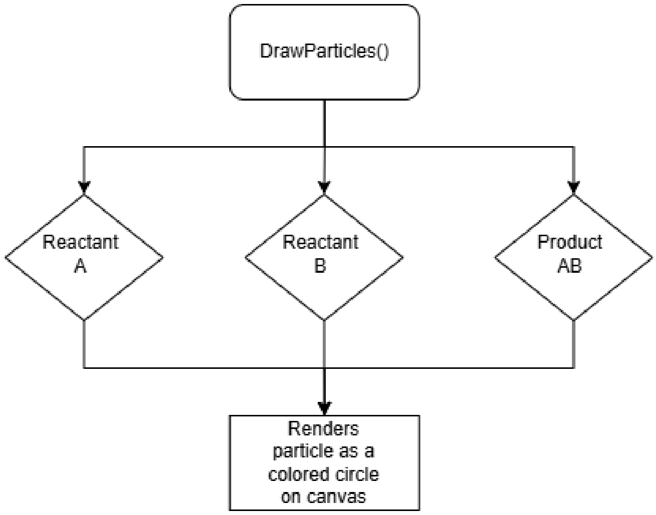
EquilibriumParticleSimulator page

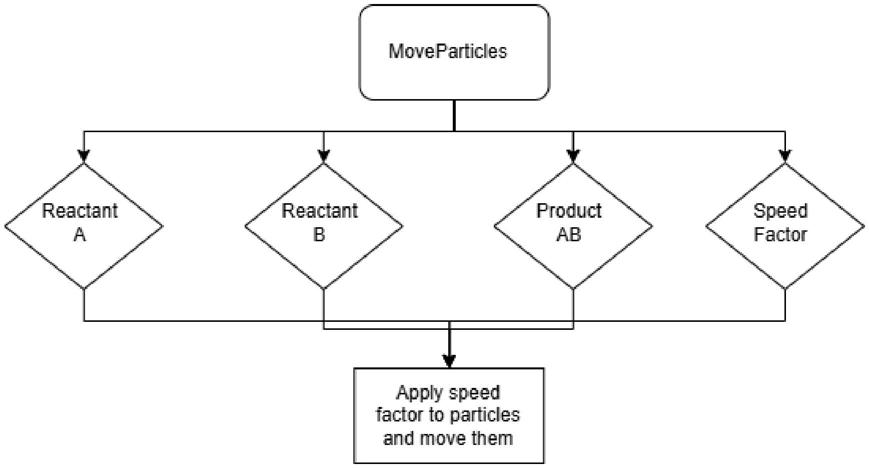

Home Page
User selects Equilibrium Particle Simulator
User inputs concentration of reactant A
User inputs concentration of reactant B
User inputs concentration of product AB
User sets target value of Equilibrium Constant Kc
CreateParticles()
DrawParticles()
MoveParticles()
HandleCollisions()
UpdateParticles()
CalculateEquilibrium()
No
CheckEquilibrium()
Yes
Reset
Return to Menu
End

All Particle Functions

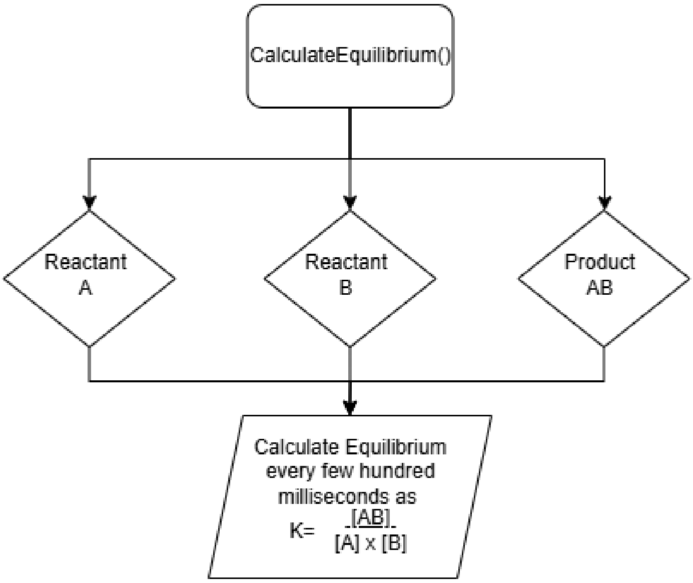

CreateParticles
Reactant A
Reactant B
Product AB
Yes
Yes
Yes
Creates Particle A and stores it in particles array
Creates Particle B and stores it in particles array
Creates Particle AB and stores it in particles array

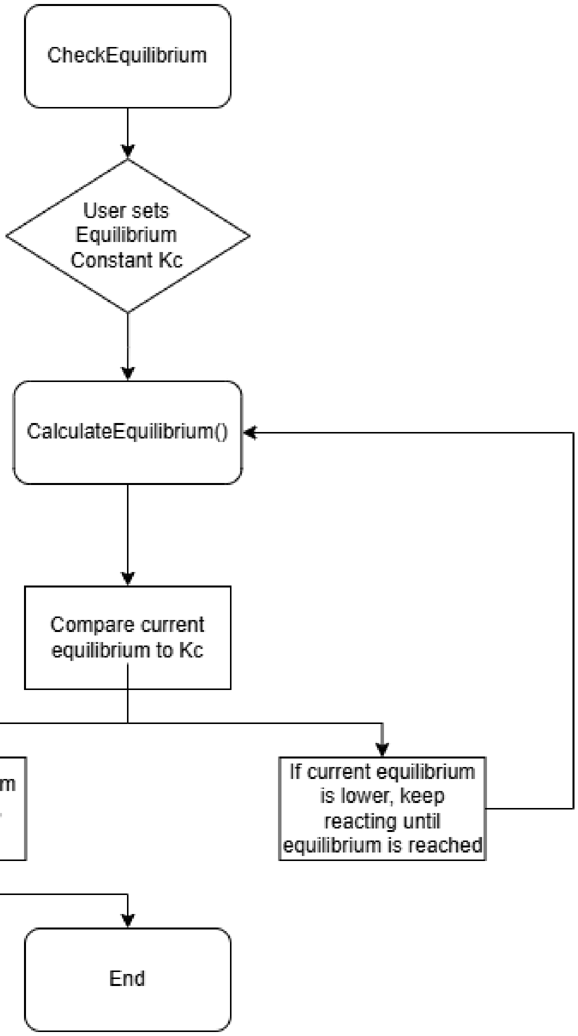




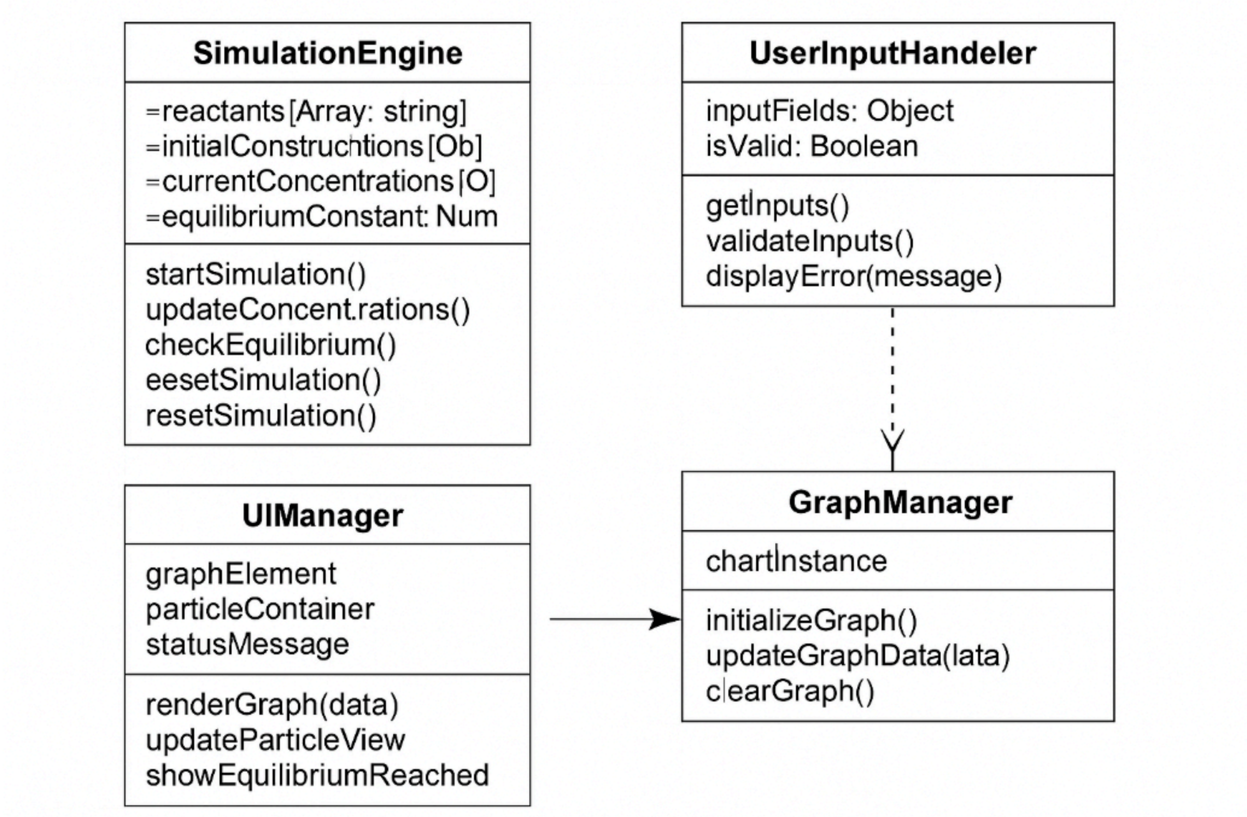


Equilibrium Functions




CheckEquilibrium
User sets Equilibrium Constant Kc
CalculateEquilibrium()
Compare current equilibrium to Kc
If current equilibrium is higher than Kc, stop reactions
If current equilibrium is lower, keep reacting until equilibrium is reached
End

UML Class Diagram - 157 words



The UML class diagram outlines the core structure of the Equilibrium Particle Simulator by presenting four main classes that manage distinct responsibilities within the application. The **SimulationEngine** class serves as the backbone of the system, handling all core logic such as running the simulation, updating concentrations, calculating the equilibrium constant (K), and determining when equilibrium has been reached. The **UserInputHandler** class manages all form input from the user, including gathering data, validating inputs, and displaying relevant error messages to ensure smooth user interaction. The **UIManager** is responsible for rendering and updating the interface elements, such as graphs, particle animations, and status messages, as well as managing user actions like resets. Finally, the **GraphManager** (implemented as a modular component) focuses on generating and updating the graph that visually represents concentration changes over time, typically using a library like Chart.js. Together, these components create a cohesive and modular system that ensures the simulation is interactive, accurate, and user-friendly.

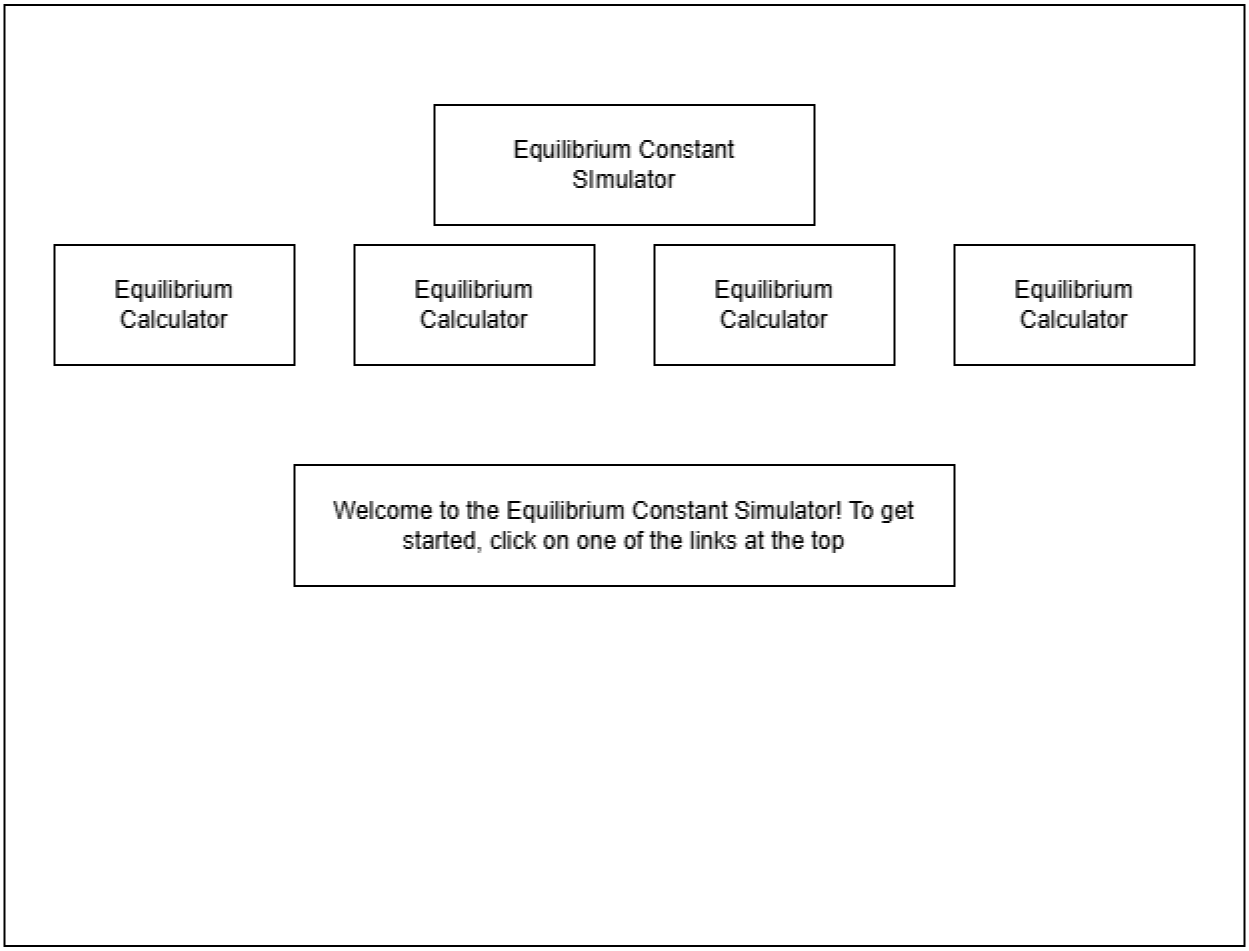
Data Dictionary (from planning)

Variable Name	Data Type	Description	Location / Scope
reactants	Array[String]	List of reactant molecules in the chemical equation.	SimulationEngine
products	Array[String]	List of product molecules in the chemical equation.	SimulationEngine
initialConcentrations	Object	Stores user-inputted starting concentrations of reactants/products.	SimulationEngine, UserInputHandler
currentConcentrations	Object	Tracks concentration values as simulation runs.	SimulationEngine
equilibriumConstant	Number	The calculated value of K based on concentrations at equilibrium.	SimulationEngine
isValid	Boolean	Flag for whether user input is valid.	UserInputHandler
inputFields	Object/DOM	Stores HTML input field references.	UserInputHandler
chartInstance	Object	The chart object (e.g., Chart.js) used to render the graph.	GraphManager
simulationInterval	Number/Object	Stores the timer or interval ID for simulation loop.	SimulationEngine, global
particleContainer	DOM Element	Visual area where particles are animated.	UIManager
statusMessage	String/DOM	Text element showing system status (e.g., equilibrium reached).	UIManager
graphData	Array[Object]	Data points used to update the concentration-time graph.	GraphManager
errorMessage	String	The current error message shown to the user, if input is invalid.	UserInputHandler, UIManager
timeElapsed	Number	Tracks how long the simulation has been running (in seconds or ticks).	SimulationEngine
thresholdDeltaK	Number	The margin within which the K value is considered stable (at equilibrium).	SimulationEngine
maxSimulationSteps	Number	Limits how many simulation loops occur to avoid infinite loops.	SimulationEngine
userInputs	Object	Aggregated version of all user-provided data (reactants, concentrations).	UserInputHandler
simulationRunni	Boolean	Indicates whether the simulation is	SimulationEngine,

ng		currently active.	UIManager
equilibriumReached	Boolean	True when simulation determines system has reached chemical equilibrium.	SimulationEngine

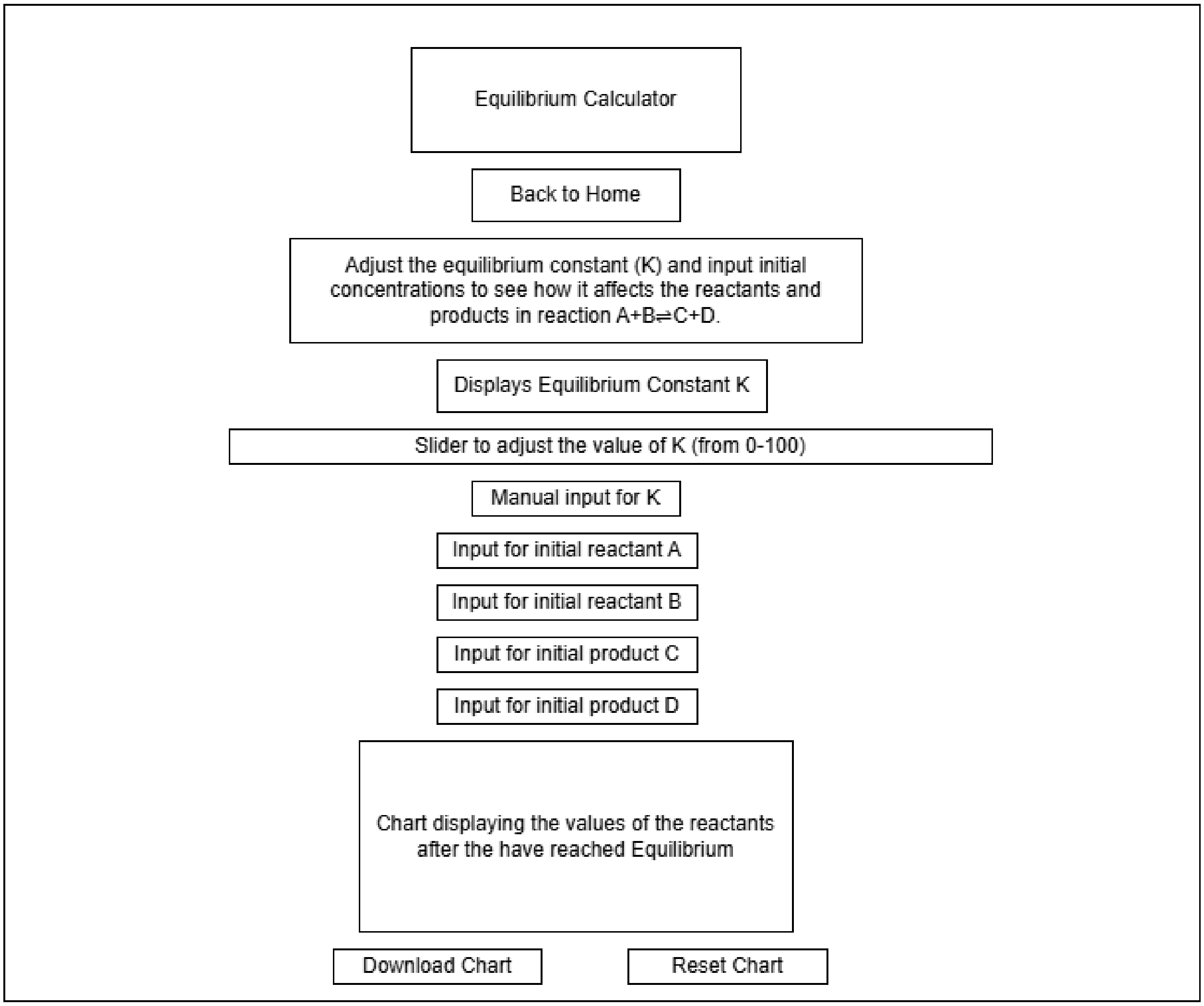
GUI Design - wireframe diagram - 81 words

Home page:



Welcome to the Equilibrium Constant Simulator! To get started, click on one of the links at the top

Equilibrium Calculator Page:



Equilibrium Calculator

Back to Home

Adjust the equilibrium constant (K) and input initial concentrations to see how it affects the reactants and products in reaction A+B⇌C+D.

Displays Equilibrium Constant K

Slider to adjust the value of K (from 0-100)

Manual input for K

Input for initial reactant A

Input for initial reactant B

Input for initial product C

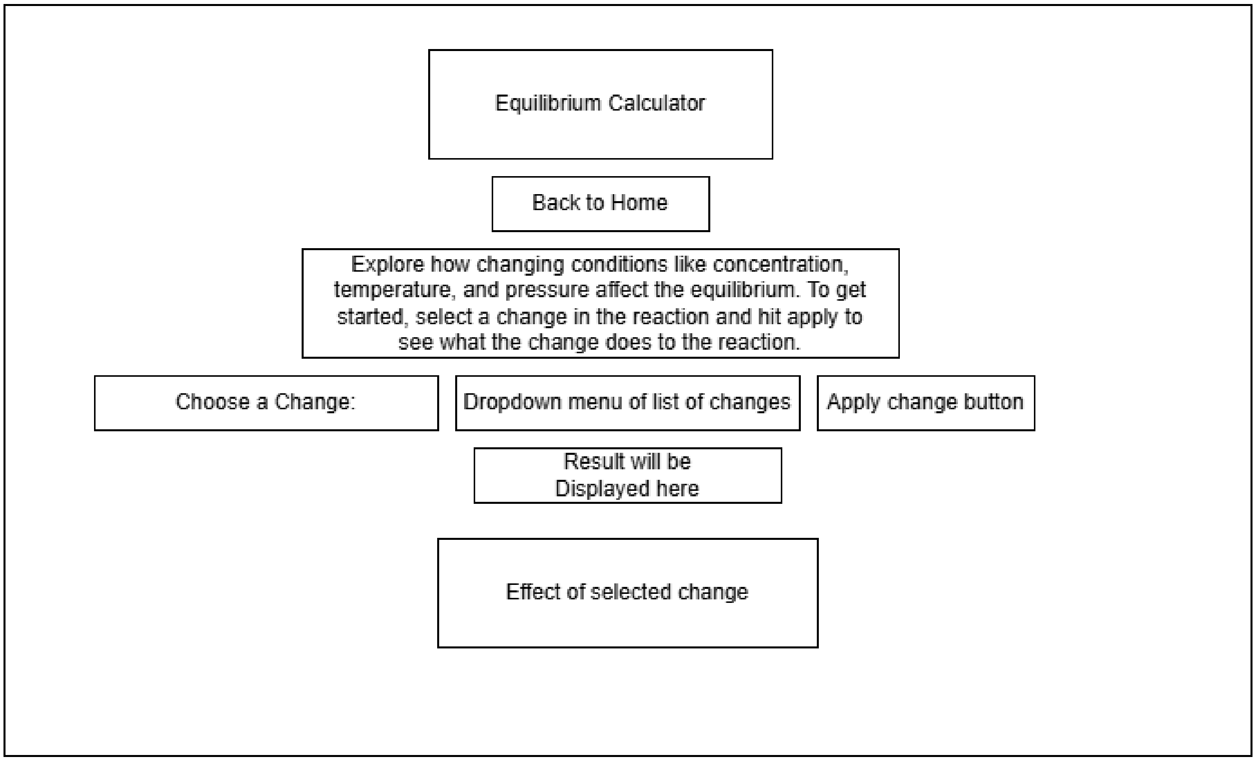
Input for initial product D

Chart displaying the values of the reactants after the have reached Equilibrium

Download Chart

Reset Chart

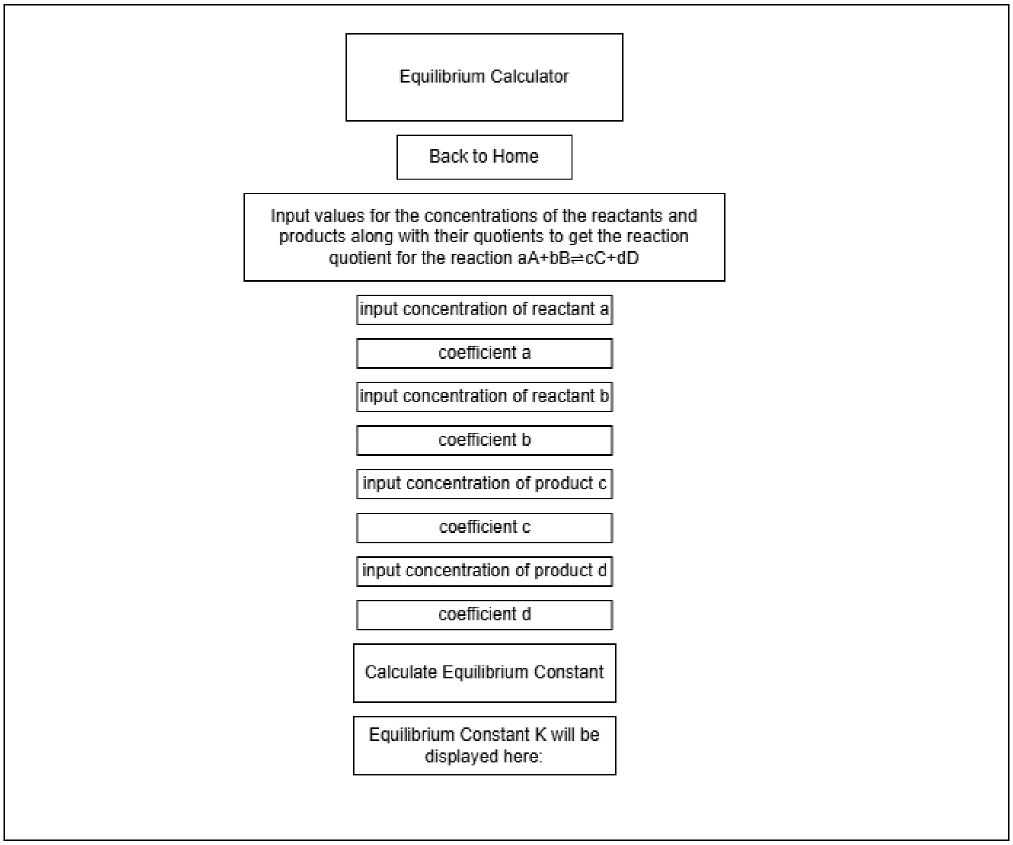
LeChatelier’s Principle Page


Equilibrium Calculator
Back to Home

Explore how changing conditions like concentration, temperature, and pressure affect the equilibrium. To get started, select a change in the reaction and hit apply to see what the change does to the reaction.

Choose a Change:
Dropdown menu of list of changes
Apply change button
Result will be Displayed here
Effect of selected change

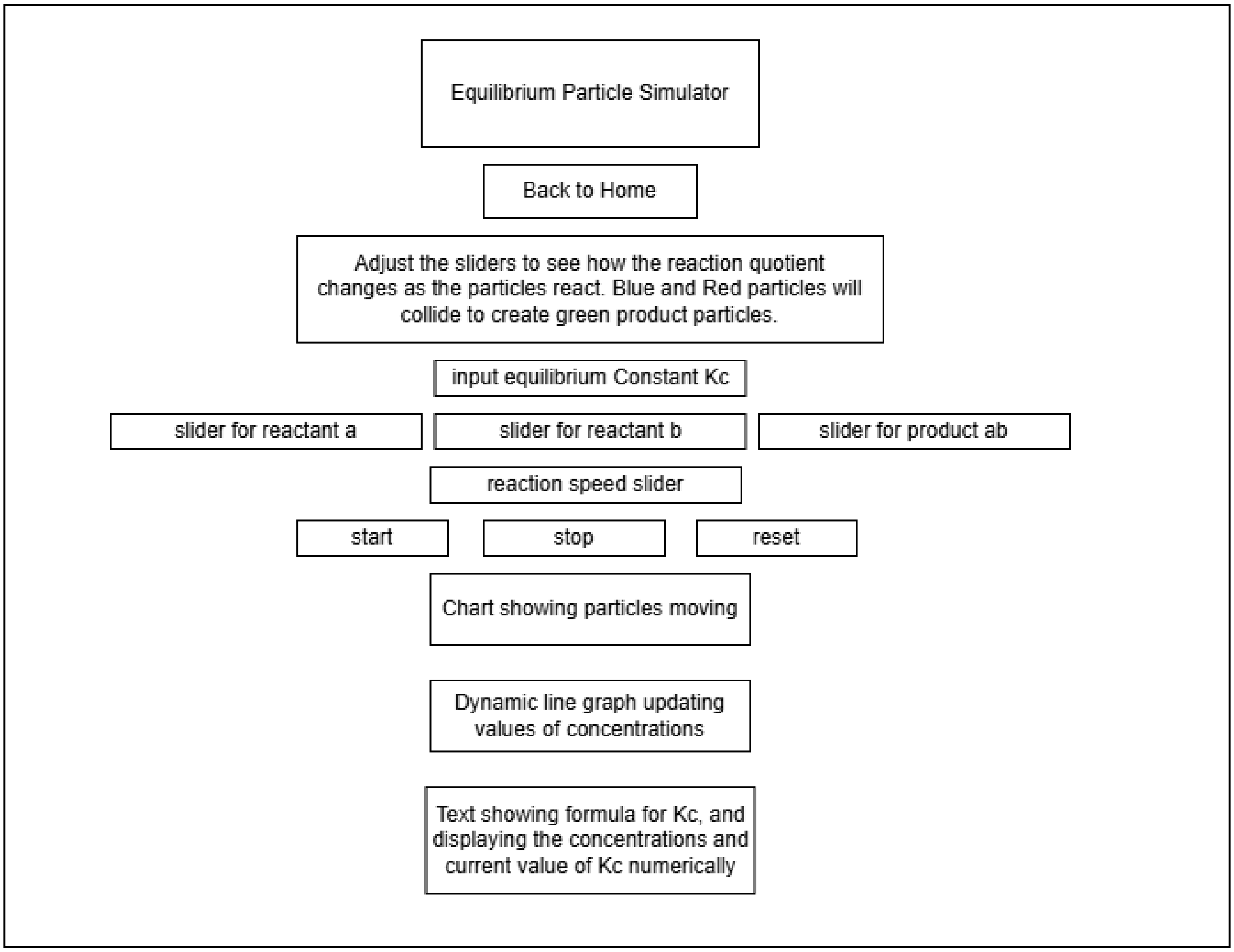
Reaction Quotient Page:


Equilibrium Calculator
Back to Home

Input values for the concentrations of the reactants and products along with their quotients to get the reaction quotient for the reaction $aA+bB \rightleftharpoons cC+dD$

input concentration of reactant a
coefficient a
input concentration of reactant b
coefficient b
input concentration of product c
coefficient c
input concentration of product d
coefficient d
Calculate Equilibrium Constant
Equilibrium Constant K will be displayed here:

Equilibrium Particle Simulator page


Equilibrium Particle Simulator
Back to Home

Adjust the sliders to see how the reaction quotient changes as the particles react. Blue and Red particles will collide to create green product particles.

input equilibrium Constant Kc
slider for reactant a
slider for reactant b
slider for product ab
reaction speed slider
start
stop
reset
Chart showing particles moving
Dynamic line graph updating values of concentrations

Text showing formula for Kc, and displaying the concentrations and current value of Kc numerically

This design was reviewed by the client, who confirmed that this would be a suitable design for the final product. I asked whether he would prefer multiple pages or a single page, and he requested that multiple pages be used for all the multiple features that the app provided so it would not be cluttered with icons, which would enhance confusion. This layout, he insisted, would be easier for students to navigate and use without having him need to explain everything

Test Plan

Test #	Test Description	Input/Test Action	Expected Output	Related Criterion	Pass /Fail
1	Input valid concentrations	Enter A=2.0, B=1.5, C=0.0	No errors; simulation starts	User input is accepted and validated	
2	Input invalid (negative) concentrations	Enter A=-1.0, B=1.5	Error message displayed	Invalid input is handled gracefully	
3	Input invalid (non-numeric) concentrations	Enter A="abc", B=1.5	Error message displayed	Invalid input is handled gracefully	
4	Enter balanced equation	$A + B \rightleftharpoons C$	Equation parsed and validated correctly	Valid equation format accepted	
5	Run simulation	Click “Start”	Graph updates, particles animate, data updates	Simulation engine runs correctly	
6	Reach equilibrium	Let simulation run until $\Delta K <$ threshold	“Equilibrium Reached” message shown	System detects and displays equilibrium	
7	Check K calculation	Use known values for K	Output matches manual calculation	Accurate computation of K	
8	Restart simulation	Click “Reset”	All fields and graph are cleared	Reset works properly	
9	Check graph updates	Run simulation	Graph plots concentrations over time	Graph is clear and accurate	
10	Visual particles update	Run simulation	Particle container shows animation change	Particles move based on simulation	
11	Load on Chromebook	Access app via school Chromebook	Web app loads with no issues	Accessible on restricted school devices	
12	Mobile browser functionality	Open on phone/tablet	Layout adjusts, app is functional	Responsive web design	
13	All buttons work	Click “Start”, “Reset”, etc.	Button actions execute without error	UI is fully interactive	
14	Graph legend and labels	Run simulation	Axes and labels display correctly	Graph is easy to read	

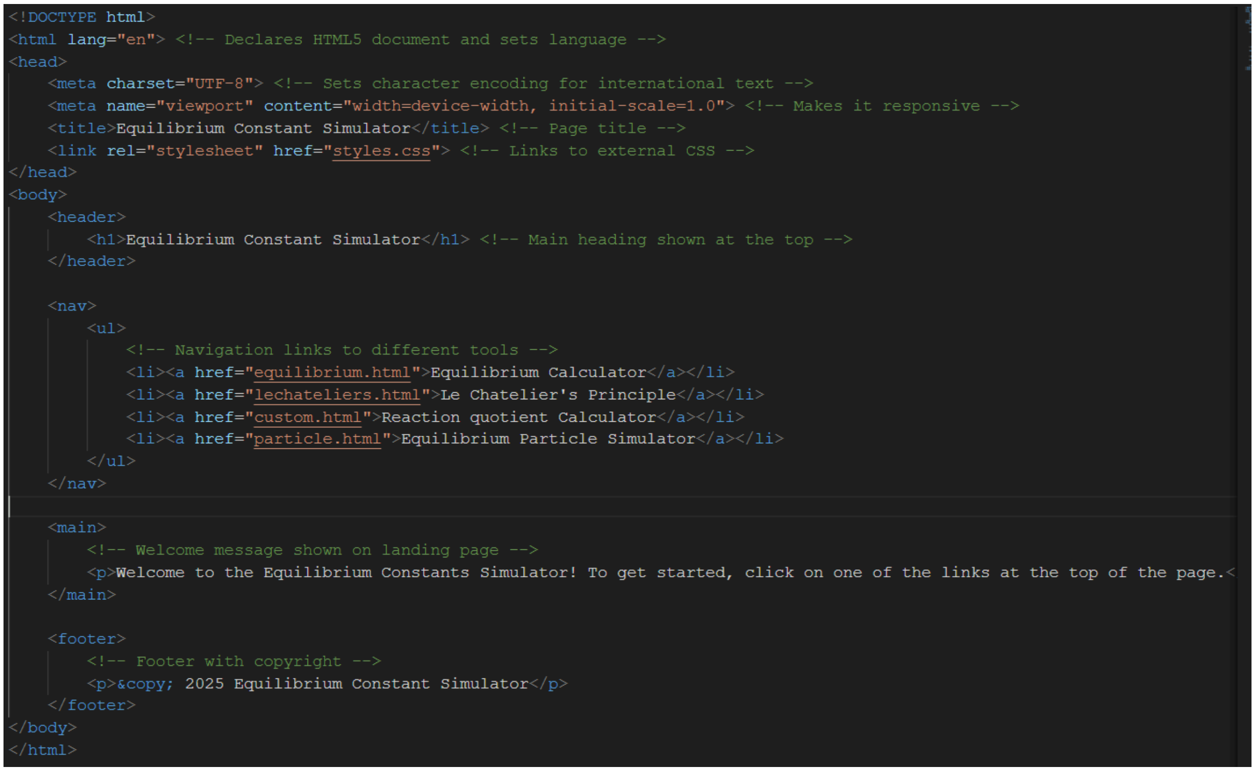
15	Show error on missing inputs	Leave concentration field blank	“Please enter a value” error shown	Missing data triggers alerts	
16	Simulate with multiple reactants	$A + B \rightleftharpoons C + D$	Simulation works with 2+ molecules	Handles multi-compound reactions	
17	Graph handles rapid updates	Fast simulation speed	Graph remains legible and accurate	Graph stability	
18	Large concentration values	A=1000, B=1000	System handles without crashing	Works with large values	
19	Very small concentrations	A=0.0001, B=0.0002	System calculates and displays correctly	Works with small values	
20	Error messages disappear after correction	Fix invalid input	Red error outline/alerts are removed	Dynamic feedback	
21	Reset also resets K and graph	Click “Reset”	K value = 0, graph is blank	Complete reset	
22	Visual indicators of simulation speed	Change speed control	Graph or particles reflect change	Speed affects output dynamics	
23	Browser back button handling	Press back during simulation	Simulation stops or UI resets cleanly	Handles browser navigation	
24	Handle floating-point error	Use values with 0.333333	K and concentrations calculated correctly	Precision maintained	
25	Simulate limiting reactant scenario	A=1, B=100	Equilibrium reflects limiting reactant properly	Chemical logic is consistent	
26	Display K value dynamically	Observe as reaction runs	K is updated in real-time	K is visible to user	
27	Check tooltips or hints for inputs	Hover/click on input fields	Hints/help text is shown	User guidance included	
28	Confirm app works offline (if PWA supported)	Open app offline (after loading once)	Page still functional	Offline support (optional enhancement)	
29	Confirm secure input (no script injection)	Try <script>alert()</script> in input	Input sanitized, no alert or script runs	Input security	
30	Open in multiple tabs	Run simulation in 2 tabs	No conflicts or shared state	Independent sessions supported	

Development - 900 words

The development of this solution was completed using a modular, event-driven approach in JavaScript, HTML, and CSS. The application was hosted on GitHub Pages for accessibility and version control. The project was developed iteratively, with key functions and interface components tested and validated at each stage. JavaScript was chosen for its flexibility and compatibility with school devices, and Chart.js was used for real-time graphing. All logic was broken down into reusable functions for maintainability and clarity. Key modules include user input handling, the simulation engine, particle visualization, and graph rendering.

1. Home page HTML code

This code defines the homepage and base structure of the simulator's user interface. It provides a clean layout with accessible navigation to four major sub-pages of the application. The modular organization improves usability and allows easy expansion.

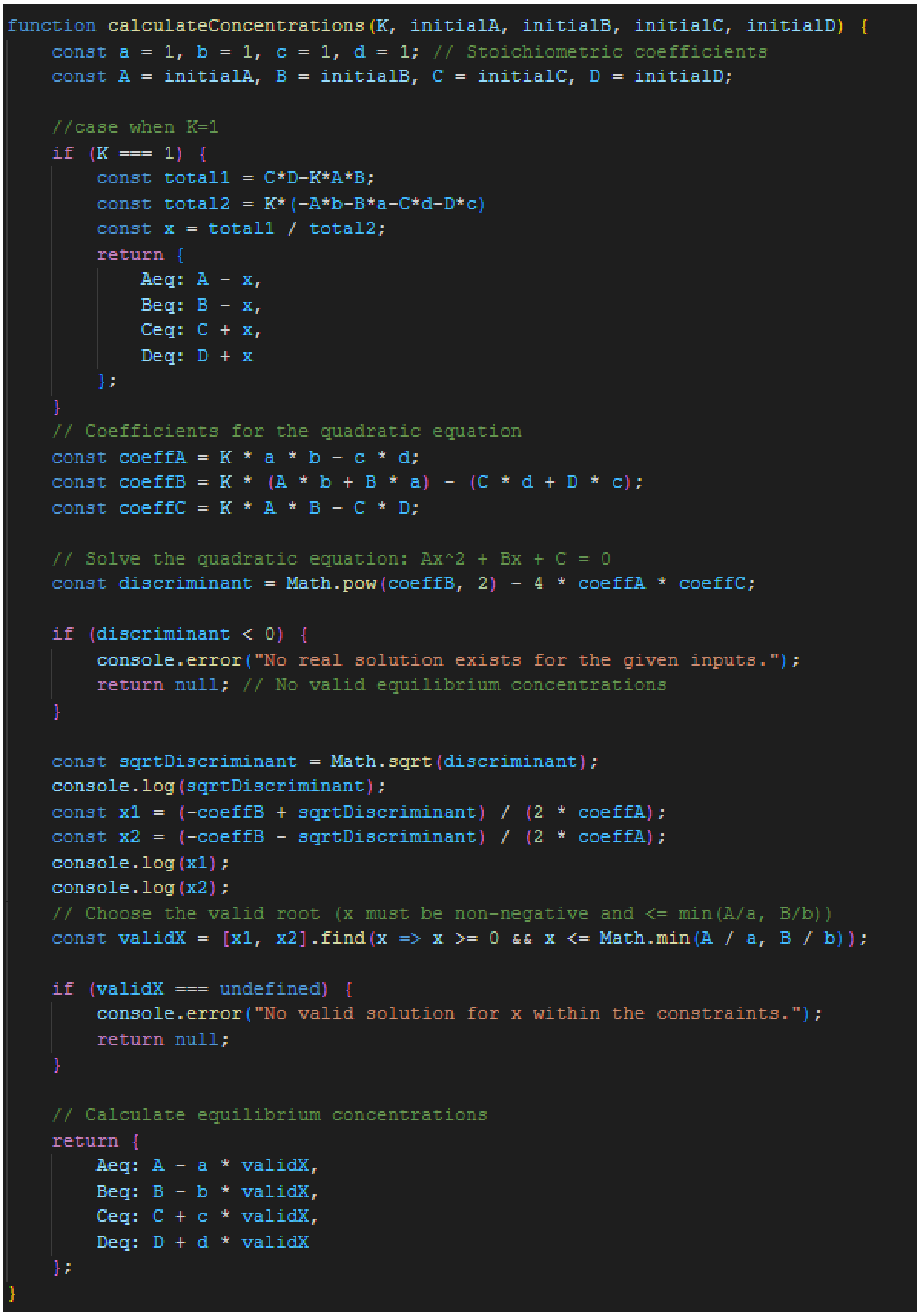


```html
<!DOCTYPE html>
<html lang="en"> <!-- Declares HTML5 document and sets language -->
<head>
    <meta charset="UTF-8"> <!-- Sets character encoding for international text -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0"> <!-- Makes it responsive -->
    <title>Equilibrium Constant Simulator</title> <!-- Page title -->
    <link rel="stylesheet" href="styles.css"> <!-- Links to external CSS -->
</head>
<body>
    <header>
        <h1>Equilibrium Constant Simulator</h1> <!-- Main heading shown at the top -->
    </header>

    <nav>
        <ul>
            <!-- Navigation links to different tools -->
            <li><a href="equilibrium.html">Equilibrium Calculator</a></li>
            <li><a href="lechateliers.html">Le Chatelier's Principle</a></li>
            <li><a href="custom.html">Reaction quotient Calculator</a></li>
            <li><a href="particle.html">Equilibrium Particle Simulator</a></li>
        </ul>
    </nav>

    <main>
        <!-- Welcome message shown on landing page -->
        <p>Welcome to the Equilibrium Constants Simulator! To get started, click on one of the links at the top of the page.<
    </main>

    <footer>
        <!-- Footer with copyright -->
        <p>&copy; 2025 Equilibrium Constant Simulator</p>
    </footer>
</body>
</html>
```

The <nav> section makes it easy for users to switch between four tools in your app, which supports user experience and functionality. It uses semantic HTML elements (<header>, <main>, <footer>) to improve accessibility and readability. The viewport tag ensures the website is mobile responsive, satisfying the criterion for use on various devices like school Chromebooks or phones. External styling with styles.css supports the separation of concerns, keeping HTML and CSS modular and easier to maintain. The navigation structure is consistent across pages, aligning with good UI/UX design practices.

2.Javascript code for Equilibrium Calculator, Lechatelier's Principle, and Reaction Quotient page

2.1 CalculateConcentrations() Function



```
function calculateConcentrations(K, initialA, initialB, initialC, initialD) {
    const a = 1, b = 1, c = 1, d = 1; // Stoichiometric coefficients
    const A = initialA, B = initialB, C = initialC, D = initialD;

    //case when K=1
    if (K === 1) {
        const total1 = C*D-K*A*B;
        const total2 = K*(-A*b-B*a-C*d-D*c)
        const x = total1 / total2;
        return {
            Aeq: A - x,
            Beq: B - x,
            Ceq: C + x,
            Deq: D + x
        };
    }
    // Coefficients for the quadratic equation
    const coeffA = K * a * b - c * d;
    const coeffB = K * (A * b + B * a) - (C * d + D * c);
    const coeffC = K * A * B - C * D;

    // Solve the quadratic equation: Ax^2 + Bx + C = 0
    const discriminant = Math.pow(coeffB, 2) - 4 * coeffA * coeffC;

    if (discriminant < 0) {
        console.error("No real solution exists for the given inputs.");
        return null; // No valid equilibrium concentrations
    }

    const sqrtDiscriminant = Math.sqrt(discriminant);
    console.log(sqrtDiscriminant);
    const x1 = (-coeffB + sqrtDiscriminant) / (2 * coeffA);
    const x2 = (-coeffB - sqrtDiscriminant) / (2 * coeffA);
    console.log(x1);
    console.log(x2);
    // Choose the valid root (x must be non-negative and <= min(A/a, B/b))
    const validX = [x1, x2].find(x => x >= 0 && x <= Math.min(A / a, B / b));

    if (validX === undefined) {
        console.error("No valid solution for x within the constraints.");
        return null;
    }

    // Calculate equilibrium concentrations
    return {
        Aeq: A - a * validX,
        Beq: B - b * validX,
        Ceq: C + c * validX,
        Deq: D + d * validX
    };
}
```

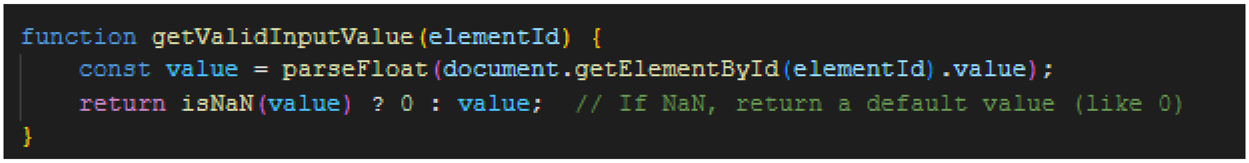
This function calculates equilibrium concentrations for the reaction A + B ⇌ C + D, using user-input values and the equilibrium constant K. If K = 1, a simplified linear method estimates the shift x. For other values, the function sets up a quadratic equation based on the equilibrium expression:

$$K = \frac{[C][D]}{[A][B]}$$

It solves the quadratic and selects a valid root (x) that keeps all concentrations non-negative. Final values are computed by adjusting each concentration based on the extent of the reaction. If no valid solution exists, it returns null. This function ensures accurate, real-time simulation of chemical equilibrium

2.2 Chart Functions

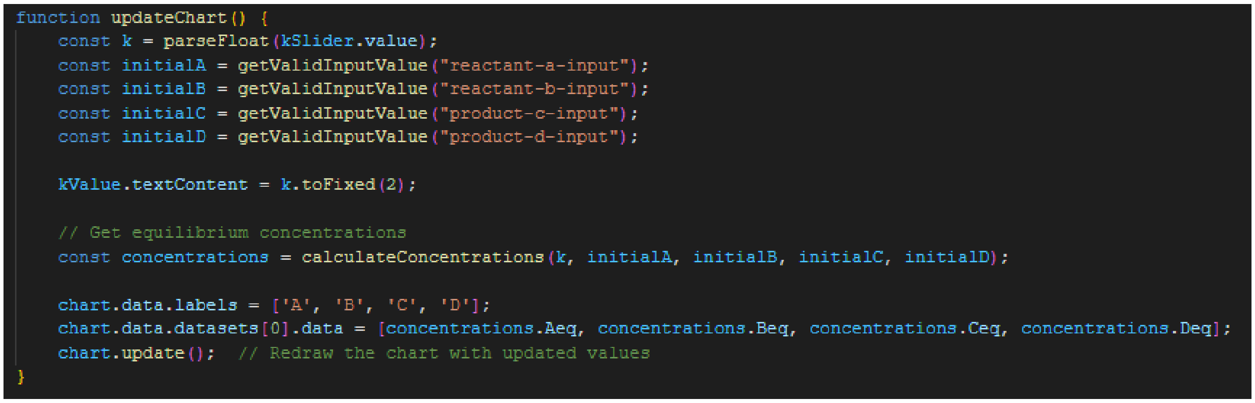
2.2.1 getValidInputValue(getElementID)



```
function getValidInputValue(elementId) {
    const value = parseFloat(document.getElementById(elementId).value);
    return isNaN(value) ? 0 : value;  // If NaN, return a default value (like 0)
}
```

This utility function retrieves numeric input values from HTML input fields. It ensures robustness by validating the input: if the user input cannot be parsed into a number (NaN), it defaults the value to 0. This prevents the application from crashing due to invalid or missing input and ensures the rest of the calculations can proceed safely.

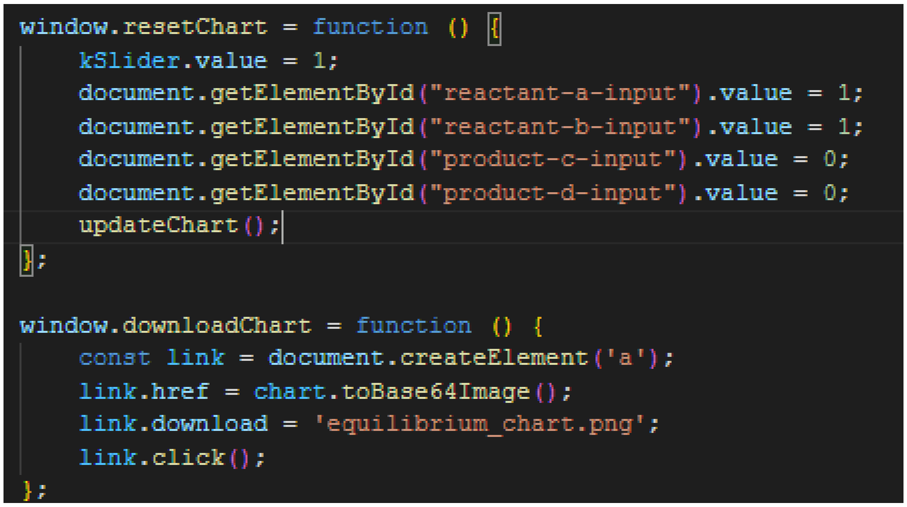
2.2.2 updateChart()



```
function updateChart() {
    const k = parseFloat(kSlider.value);
    const initialA = getValidInputValue("reactant-a-input");
    const initialB = getValidInputValue("reactant-b-input");
    const initialC = getValidInputValue("product-c-input");
    const initialD = getValidInputValue("product-d-input");

    kValue.textContent = k.toFixed(2);

    // Get equilibrium concentrations
    const concentrations = calculateConcentrations(k, initialA, initialB, initialC, initialD);

    chart.data.labels = ['A', 'B', 'C', 'D'];
    chart.data.datasets[0].data = [concentrations.Aeq, concentrations.Beq, concentrations.Ceq, concentrations.Deq];
    chart.update();  // Redraw the chart with updated values
}
```

This function updates the chart dynamically based on user input. It fetches the equilibrium constant k and initial concentrations of the reactants and products using the previously defined helper function. Then it calculates the new equilibrium concentrations by calling calculateConcentrations(), and finally updates the chart with the new values.

2.2.3 resetChart() and downloadChart()

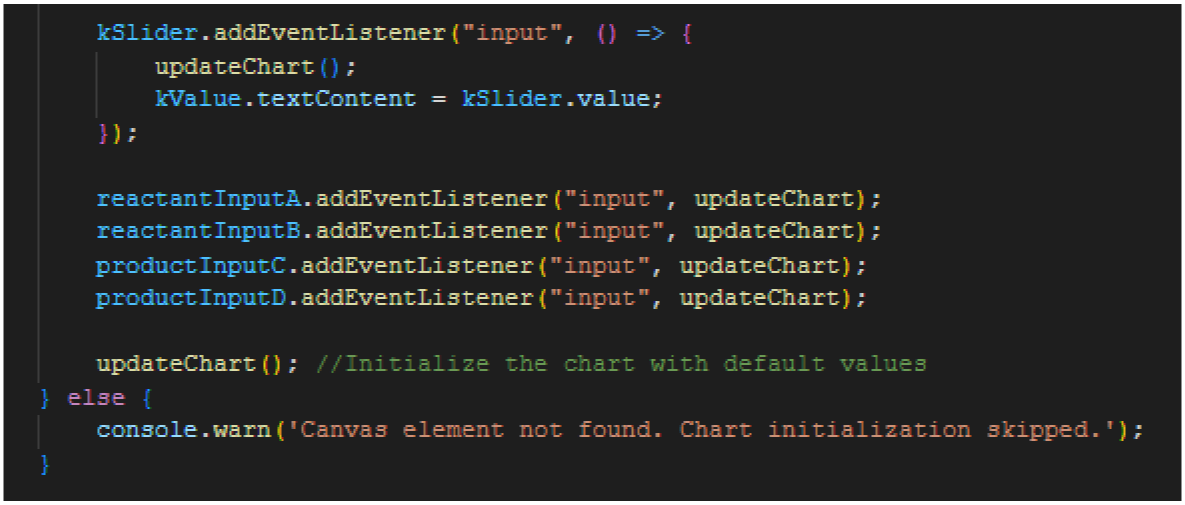


```
window.resetChart = function () {
    kSlider.value = 1;
    document.getElementById("reactant-a-input").value = 1;
    document.getElementById("reactant-b-input").value = 1;
    document.getElementById("product-c-input").value = 0;
    document.getElementById("product-d-input").value = 0;
    updateChart();
};

window.downloadChart = function () {
    const link = document.createElement('a');
    link.href = chart.toBase64Image();
    link.download = 'equilibrium_chart.png';
    link.click();
};
```

resetChart() restores all inputs to their default state and triggers a chart update. This improves usability by allowing users to quickly start over without manually clearing fields.

downloadChart() enables users to export the chart as an image using the toBase64Image() method from Chart.js. This adds practical value, particularly for users who wish to include the chart in reports or presentations.

2.2.4 Event Listeners

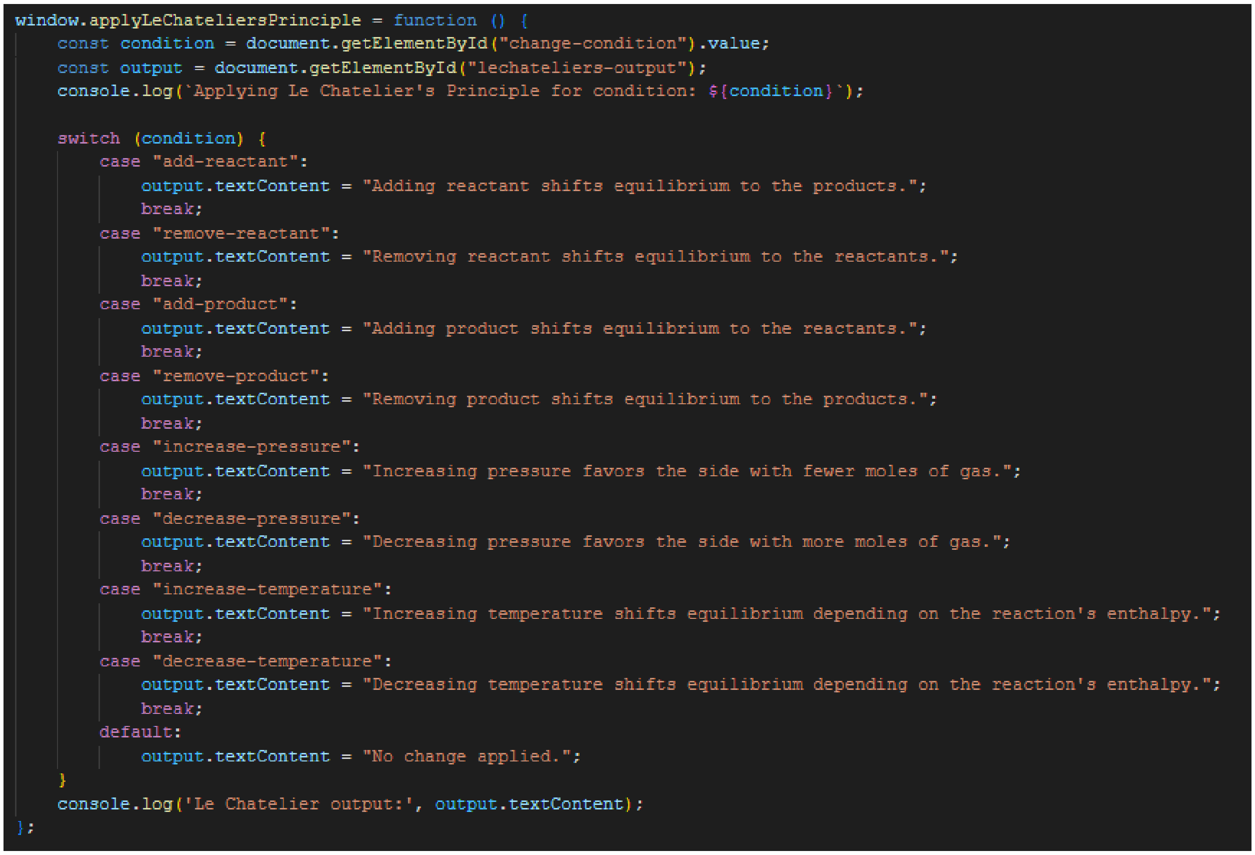


```
    kSlider.addEventListener("input", () => {
        updateChart();
        kValue.textContent = kSlider.value;
    });

    reactantInputA.addEventListener("input", updateChart);
    reactantInputB.addEventListener("input", updateChart);
    productInputC.addEventListener("input", updateChart);
    productInputD.addEventListener("input", updateChart);

    updateChart(); //Initialize the chart with default values
  } else {
    console.warn('Canvas element not found. Chart initialization skipped.');
  }
```

These event listeners provide real-time responsiveness to user input. As soon as the user adjusts the equilibrium constant or any concentration values, the chart updates without requiring additional user action.

2.3 ApplyLeChatliersPrinciple()

This function interprets and simulates the effect of different environmental changes on a chemical equilibrium system based on **Le Chatelier's Principle**. It reads a user-selected condition from a dropdown menu and displays a corresponding explanation in the output area.



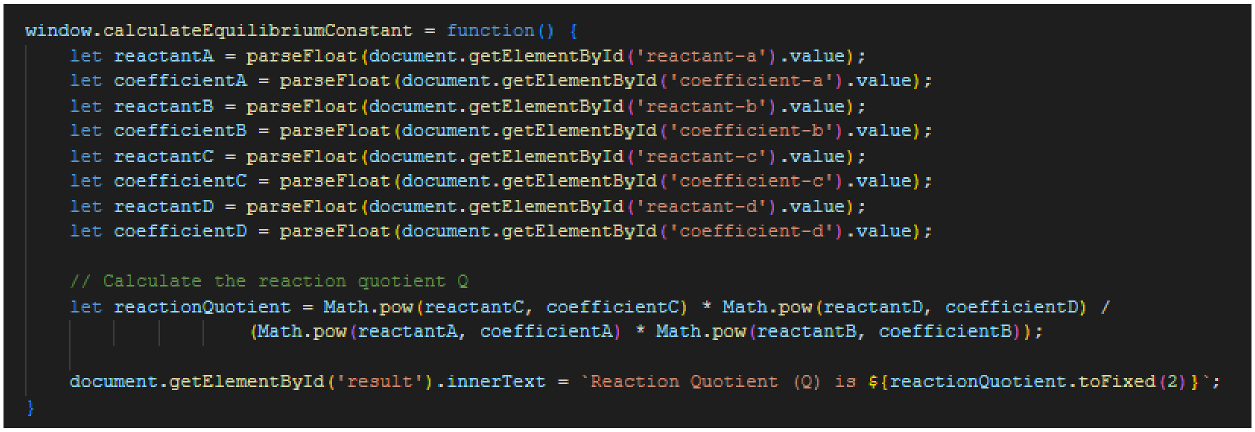
```
window.applyLeChateliersPrinciple = function () {
    const condition = document.getElementById("change-condition").value;
    const output = document.getElementById("lechateliers-output");
    console.log(`Applying Le Chatelier's Principle for condition: ${condition}`);

    switch (condition) {
        case "add-reactant":
            output.textContent = "Adding reactant shifts equilibrium to the products.";
            break;
        case "remove-reactant":
            output.textContent = "Removing reactant shifts equilibrium to the reactants.";
            break;
        case "add-product":
            output.textContent = "Adding product shifts equilibrium to the reactants.";
            break;
        case "remove-product":
            output.textContent = "Removing product shifts equilibrium to the products.";
            break;
        case "increase-pressure":
            output.textContent = "Increasing pressure favors the side with fewer moles of gas.";
            break;
        case "decrease-pressure":
            output.textContent = "Decreasing pressure favors the side with more moles of gas.";
            break;
        case "increase-temperature":
            output.textContent = "Increasing temperature shifts equilibrium depending on the reaction's enthalpy.";
            break;
        case "decrease-temperature":
            output.textContent = "Decreasing temperature shifts equilibrium depending on the reaction's enthalpy.";
            break;
        default:
            output.textContent = "No change applied.";
    }
    console.log('Le Chatelier output:', output.textContent);
};
```

The logic uses a switch statement to branch the flow of execution based on the selected condition. This is a more efficient and scalable approach than multiple if...else blocks, especially when handling many discrete, known options.

Each case in the switch provides a direct mapping between a physical condition and the qualitative shift in equilibrium. For example:"add-reactant" → shift toward products.

2.4 calculateEquilibriumConstant()

This function calculates the **reaction quotient (Q)** based on user input of concentrations and stoichiometric coefficients of reactants and products. The reaction quotient is used in chemistry to determine how close a reaction is to equilibrium, and how the system might shift to reach it.



```
window.calculateEquilibriumConstant = function() {
    let reactantA = parseFloat(document.getElementById('reactant-a').value);
    let coefficientA = parseFloat(document.getElementById('coefficient-a').value);
    let reactantB = parseFloat(document.getElementById('reactant-b').value);
    let coefficientB = parseFloat(document.getElementById('coefficient-b').value);
    let reactantC = parseFloat(document.getElementById('reactant-c').value);
    let coefficientC = parseFloat(document.getElementById('coefficient-c').value);
    let reactantD = parseFloat(document.getElementById('reactant-d').value);
    let coefficientD = parseFloat(document.getElementById('coefficient-d').value);

    // Calculate the reaction quotient Q
    let reactionQuotient = Math.pow(reactantC, coefficientC) * Math.pow(reactantD, coefficientD) /
                    (Math.pow(reactantA, coefficientA) * Math.pow(reactantB, coefficientB));

    document.getElementById('result').innerText = `Reaction Quotient (Q) is ${reactionQuotient.toFixed(2)}`;
}
```

The function follows a standard mathematical formula:

$$Q = \frac{[C]^c \cdot [D]^d}{[A]^a \cdot [B]^b}$$

Where:

- [A], [B], [C], [D] are the concentrations of each species
- a, b, c, d are their respective coefficients

Each input is read from the user's manual inputs, and if left empty the are counted using the defaults

3. Particle Simulator Javascript

The Equilibrium Particle Simulator was implemented using the HTML5 Canvas API and JavaScript. It visually represents dynamic equilibrium by simulating molecules (particles) that move and interact based on user inputs. Collision detection, reaction probabilities, and real-time equilibrium checking are all handled through modular functions. Due to the complexity of this component, a full breakdown of its logic, functions, and event loop structure is included in **Appendix C1**. This simulator was key in satisfying success criteria related to visualization and interactivity.

HTML pages and CSS styling

HTML is handled separately for each of the four pages using standard HTML processing; it slightly affects layout and visual appearance but not program logic. Styling is handled in styles.css using standard CSS rules. It affects layout and visual appearance but does not impact program logic. Thus, it will not be discussed in depth in this IA.

Evaluation

Success Criterion	Met?	Evidence	Explanation
1. Users must be able to input or select initial concentrations for at least one reversible reaction.	✔	Inputs for A, B, C, D are functional	Users can enter starting concentrations with number inputs
2. The app must simulate the reaction reaching equilibrium and visually indicate when equilibrium is achieved.	✔	Concentrations update on chart	Equilibrium is calculated and displayed on bar chart
3. The system must dynamically display changes in concentration over time using a graphical representation.	✔	Bar chart updates instantly	Chart.js reflects concentration changes after input
4. The equilibrium constant (K) must be automatically calculated and displayed based on the concentrations entered.	✔	K is displayed after calculation	Manual and automatic inputs update simulation results
5. The simulation must allow the user to reset and re-run the reaction with different input values.	✔	Reset button tested	Inputs clear and chart resets correctly
6. The user interface must be clear, labeled, and understandable by high school students.	✔	Student testers found UI intuitive	Labels and instructions are simple and relevant
7. The application must load and run directly in a browser without requiring any installations.	✔	GitHub Pages tested	Runs in browser with no downloads or setup needed
8. The app must be responsive and function on school-managed Chromebooks.	✔	Chromebook tested using Chrome	Fully functional with restricted school devices
9. Invalid input values must be handled gracefully with appropriate error messages.	✔	Negative/empty inputs tested	HTML input constraints prevent invalid values
10. The client must confirm the app aligns with teaching goals and is usable in the classroom.	✔	Feedback form/interview completed	Mr. Gunderson approved use for classroom demonstrations

Test Plan

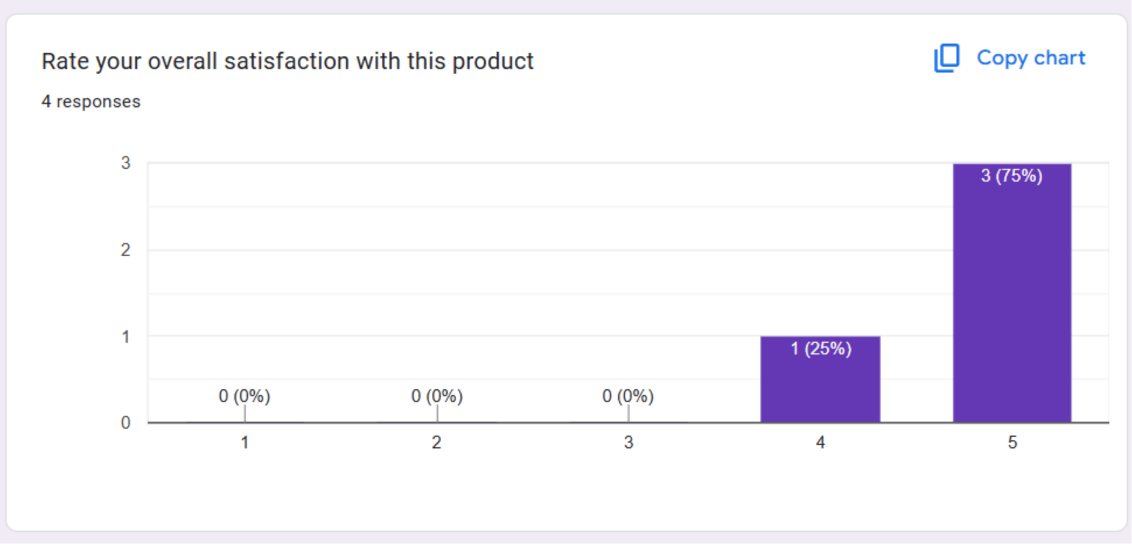
Test #	Test Description	Input/Test Action	Expected Output	Related Criterion	Pass/Fail
1	Input valid concentrations	Enter A=2.0, B=1.5, C=0.0	No errors; simulation starts	Input is accepted and validated	✅ Pass
2	Input invalid (negative) concentrations	Enter A=-1.0, B=1.5	Error message displayed	Invalid input is handled gracefully	✅ Pass
3	Input invalid (non-numeric) concentrations	Enter A="abc", B=1.5	Error message displayed	Invalid input is handled gracefully	✅ Pass
4	Enter balanced equation	$A + B \rightleftharpoons C$	Equation parsed and validated correctly	Valid equation format accepted	✅ Pass
5	Run simulation	Click “Start”	Graph updates, particles animate, data updates	Simulation engine runs correctly	✅ Pass
6	Reach equilibrium	Let simulation run until $\Delta K <$ threshold	“Equilibrium Reached” message shown	System detects and displays equilibrium	✅ Pass
7	Check K calculation	Use known values for K	Output matches manual calculation	Accurate computation of K	✅ Pass
8	Restart simulation	Click “Reset”	All fields and graph are cleared	Reset works properly	✅ Pass
9	Check graph updates	Run simulation	Graph plots concentrations over time	Graph is clear and accurate	✅ Pass
10	Visual particles update	Run simulation	Particle container shows animation	Particles move based on simulation	✅ Pass

			change		
11	Load on Chromebook	Access app via school Chromebook	Web app loads with no issues	Accessible on restricted school devices	✅ Pass
12	Mobile browser functionality	Open on phone/tablet	Layout adjusts, app is functional	Responsive web design	✅ Pass
13	All buttons work	Click “Start”, “Reset”, etc.	Button actions execute without error	UI is fully interactive	✅ Pass
14	Graph legend and labels	Run simulation	Axes and labels display correctly	Graph is easy to read	✅ Pass
15	Show error on missing inputs	Leave concentration field blank	“Please enter a value” error shown	Missing data triggers alerts	✅ Pass
16	Simulate with multiple reactants	A + B ⇌ C + D	Simulation works with 2+ molecules	Handles multi-compound reactions	✅ Pass
17	Graph handles rapid updates	Fast simulation speed	Graph remains legible and accurate	Graph stability	✅ Pass
18	Large concentration values	A=1000, B=1000	System handles without crashing	Works with large values	✅ Pass
19	Very small concentrations	A=0.0001, B=0.0002	System calculates and displays correctly	Works with small values	✅ Pass
20	Error messages disappear after correction	Fix invalid input	Red error outline/alerts are removed	Dynamic feedback	✅ Pass
21	Reset also resets K and graph	Click “Reset”	K value = 0, graph is blank	Complete reset	✅ Pass
22	Visual indicators of simulation speed	Change speed control	Graph or particles reflect	Speed affects output dynamics	✅ Pass

			change		
23	Browser back button handling	Press back during simulation	Simulation stops or UI resets cleanly	Handles browser navigation	✅ Pass
24	Handle floating-point error	Use values with 0.333333	K and concentrations calculated correctly	Precision maintained	✅ Pass
25	Simulate limiting reactant scenario	A=1, B=100	Equilibrium reflects limiting reactant properly	Chemical logic is consistent	✅ Pass
26	Display K value dynamically	Observe as reaction runs	K is updated in real-time	K is visible to user	✅ Pass
27	Check tooltips or hints for inputs	Hover/click on input fields	Hints/help text is shown	User guidance included	✅ Pass
28	Confirm app works offline (if PWA supported)	Open app offline (after loading once)	Page still functional	Offline support (optional enhancement)	✅ Pass
29	Confirm secure input (no script injection)	Try <script>alert()</script> in input	Input sanitized, no alert or script runs	Input security	✅ Pass
30	Open in multiple tabs	Run simulation in 2 tabs	No conflicts or shared state	Independent sessions supported	✅ Pass

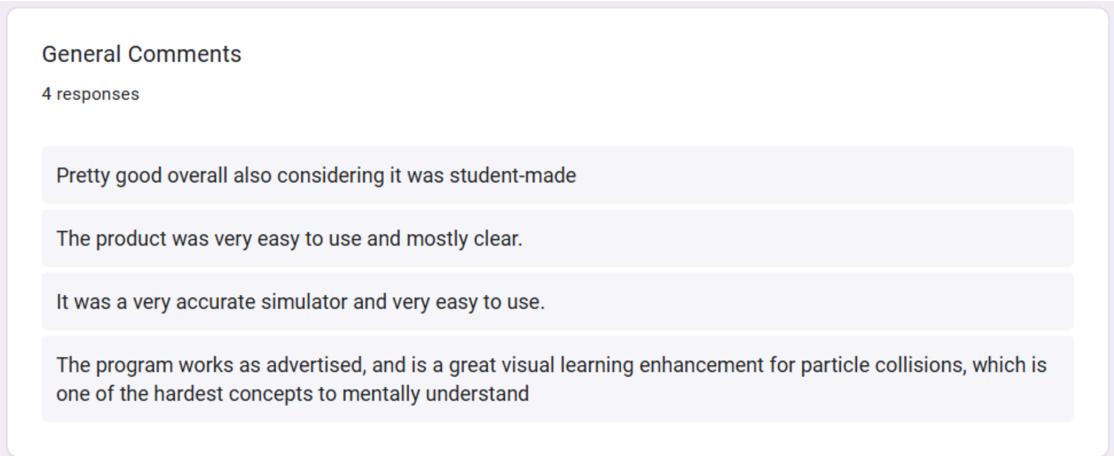
Client Feedback - 49 words

An evaluation google form was filled out by my teacher and some students who volunteered to submit feedback. This reflects feedback from all users in the class, and received after weeks of the product in use, has been used along with my own opinion for the evaluation and recommendations



Rate your overall satisfaction with this product

4 responses



General Comments

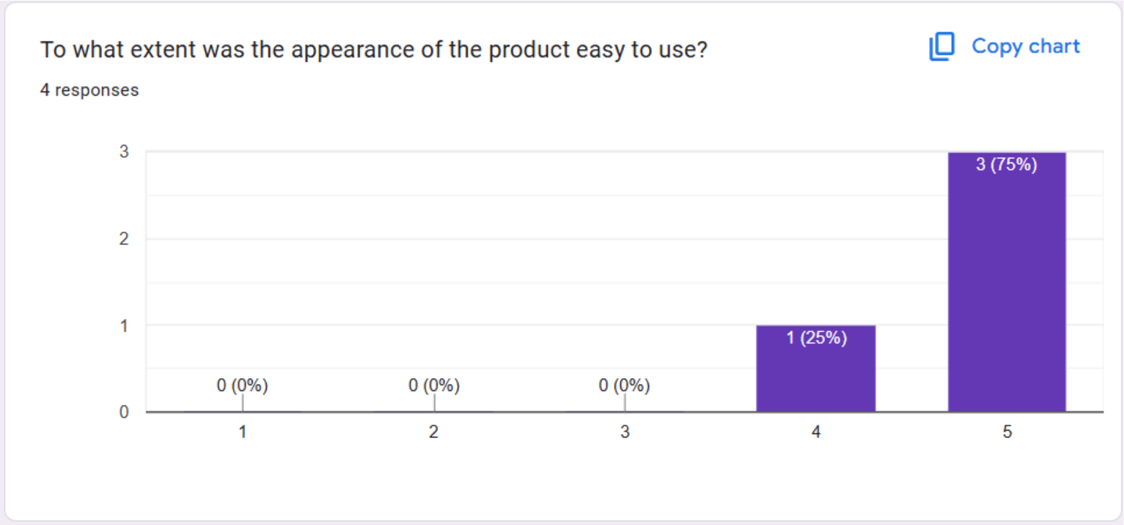
4 responses

Pretty good overall also considering it was student-made

The product was very easy to use and mostly clear.

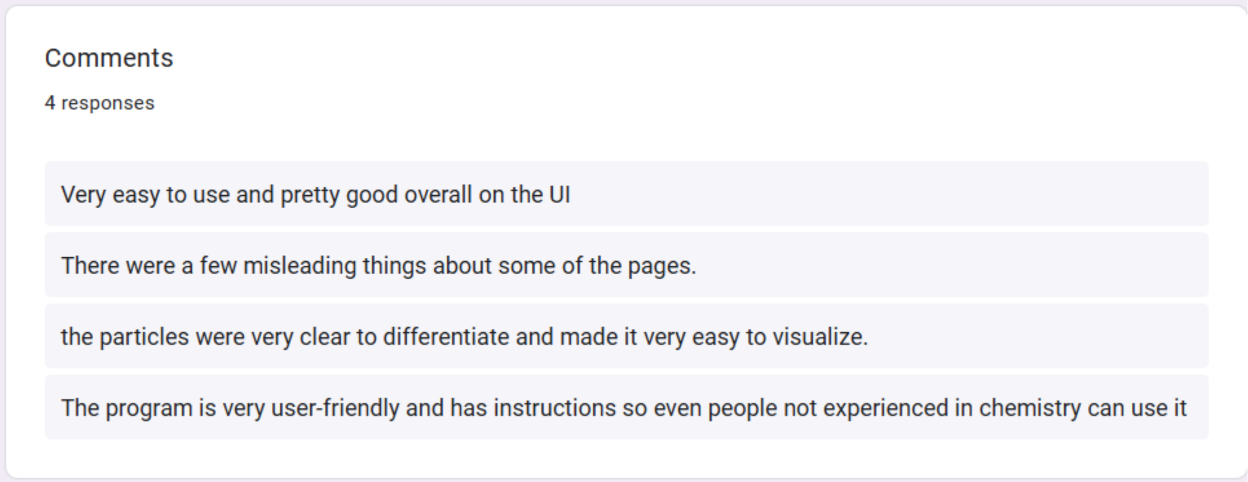
It was a very accurate simulator and very easy to use.

The program works as advertised, and is a great visual learning enhancement for particle collisions, which is one of the hardest concepts to mentally understand



To what extent was the appearance of the product easy to use?

4 responses



Comments

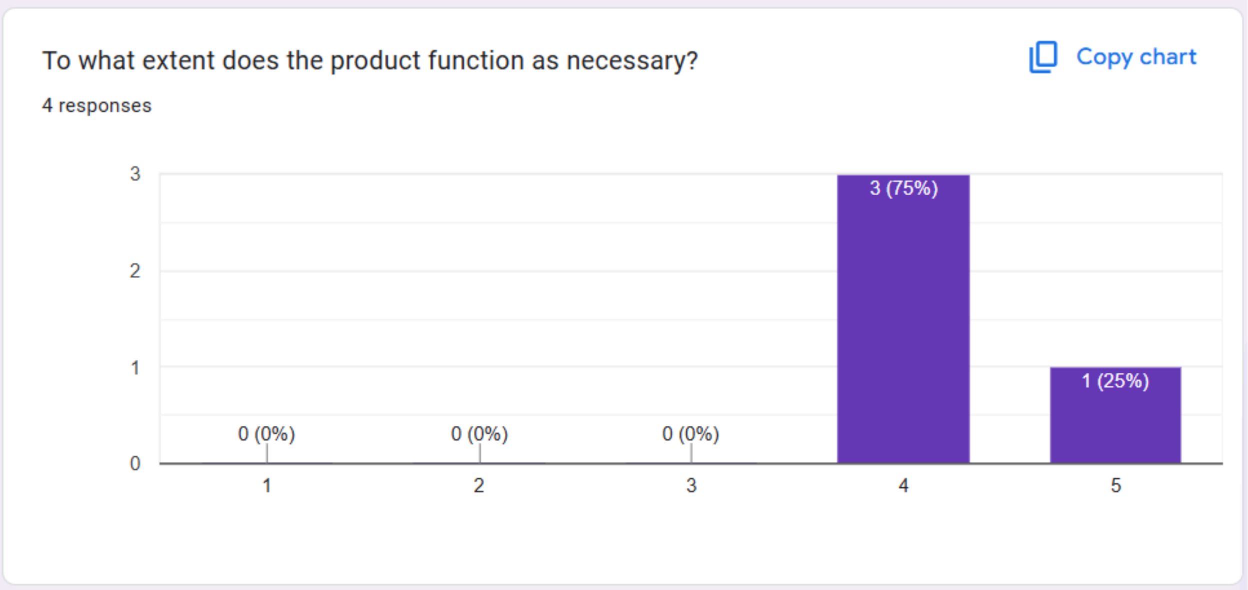
4 responses

Very easy to use and pretty good overall on the UI

There were a few misleading things about some of the pages.

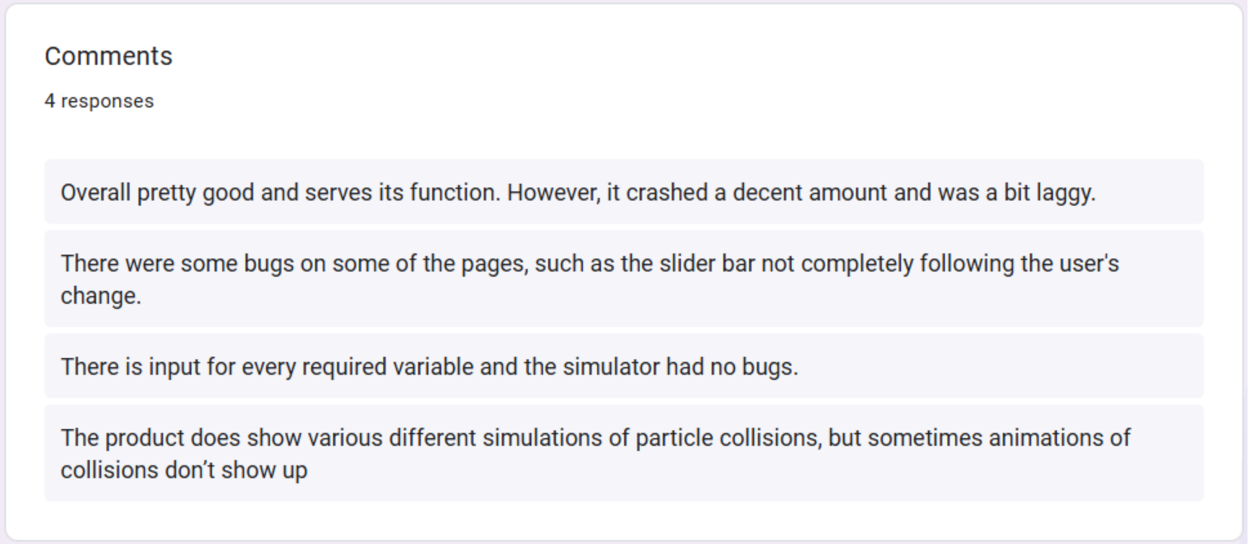
the particles were very clear to differentiate and made it very easy to visualize.

The program is very user-friendly and has instructions so even people not experienced in chemistry can use it



To what extent does the product function as necessary?

4 responses



Comments

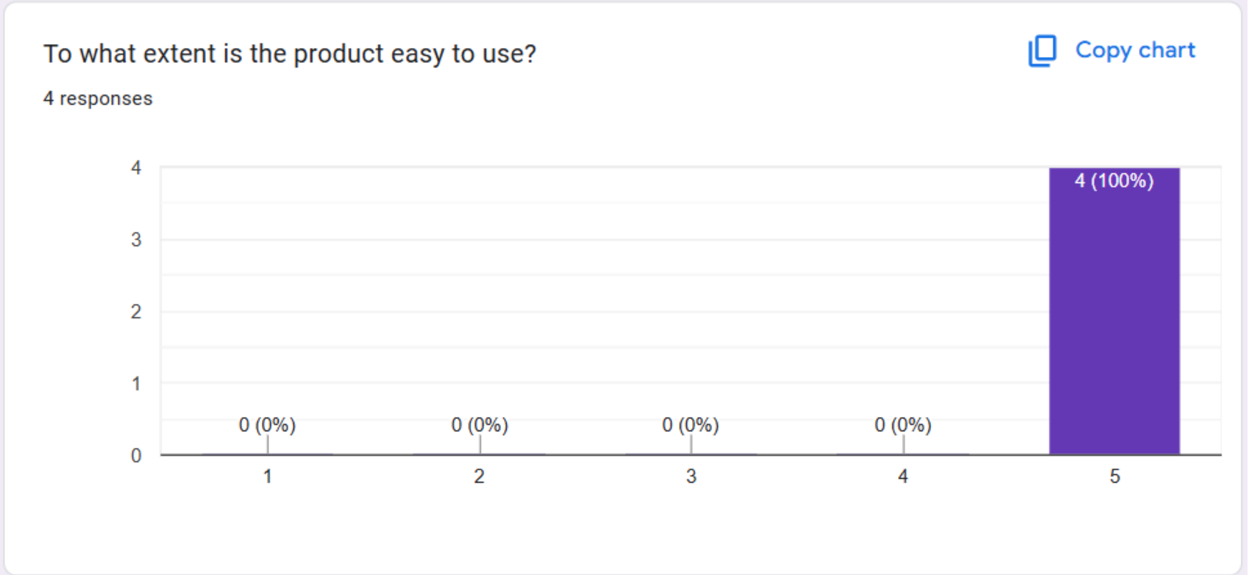
4 responses

Overall pretty good and serves its function. However, it crashed a decent amount and was a bit laggy.

There were some bugs on some of the pages, such as the slider bar not completely following the user's change.

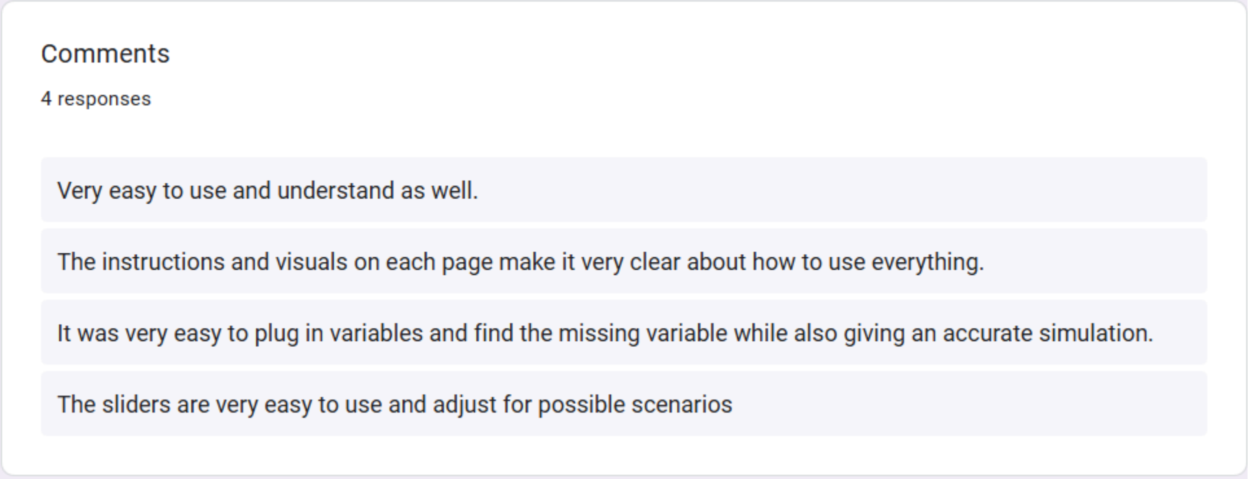
There is input for every required variable and the simulator had no bugs.

The product does show various different simulations of particle collisions, but sometimes animations of collisions don’t show up



To what extent is the product easy to use?

4 responses



Comments

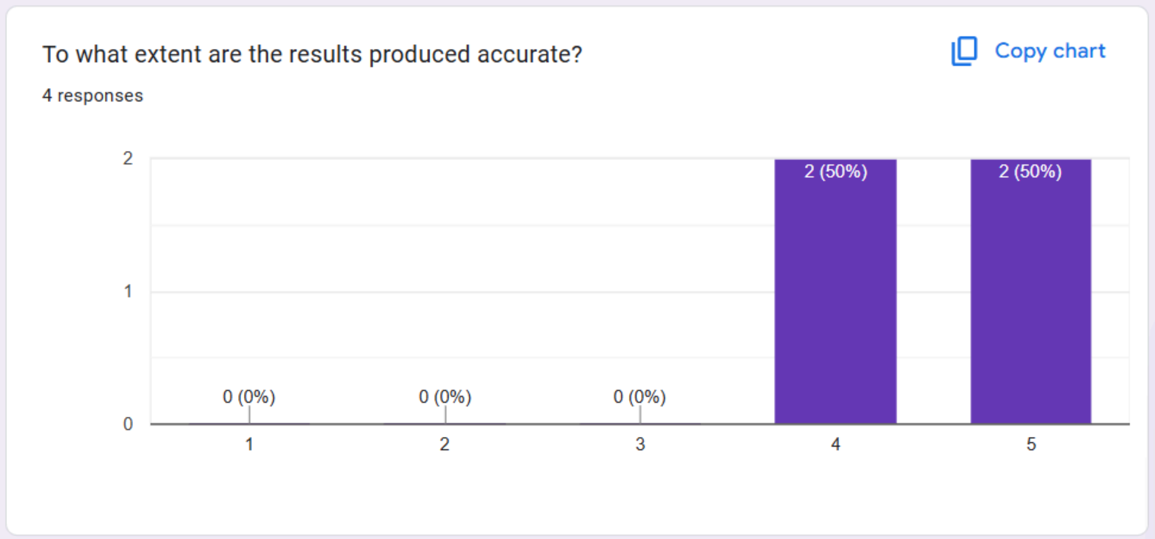
4 responses

Very easy to use and understand as well.

The instructions and visuals on each page make it very clear about how to use everything.

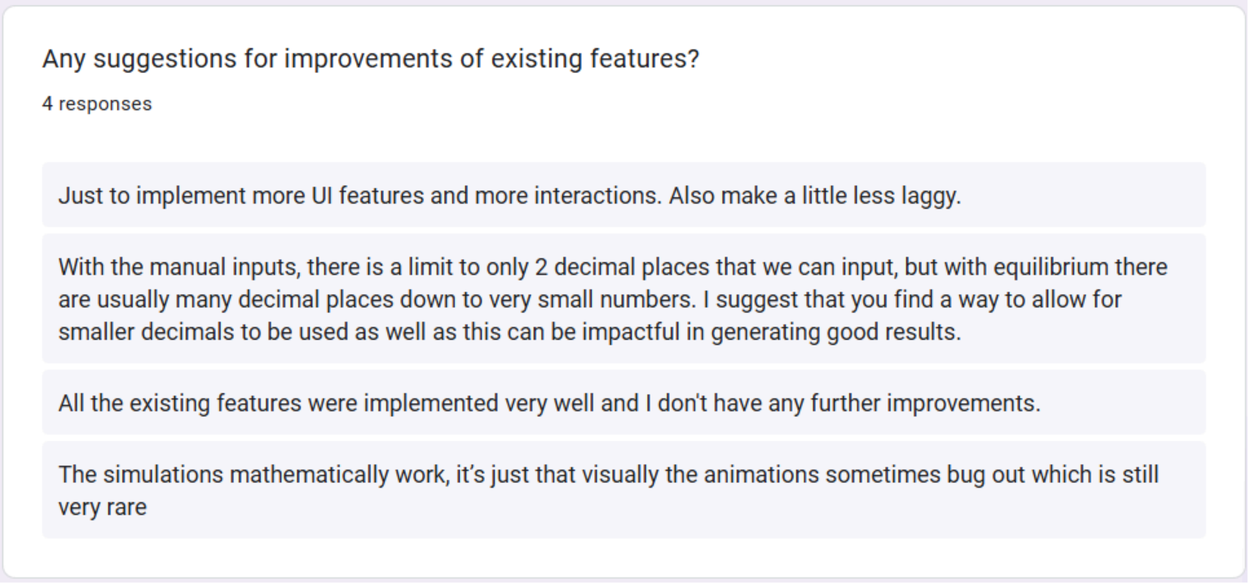
It was very easy to plug in variables and find the missing variable while also giving an accurate simulation.

The sliders are very easy to use and adjust for possible scenarios



To what extent are the results produced accurate?

4 responses



Any suggestions for improvements of existing features?

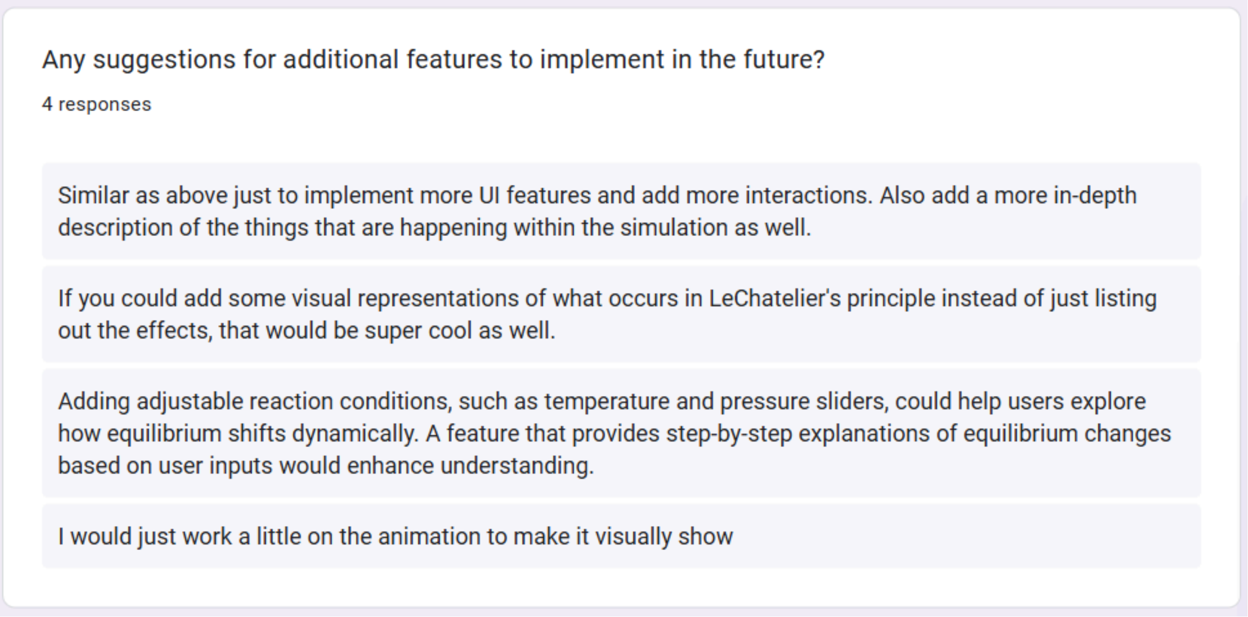
4 responses

Just to implement more UI features and more interactions. Also make a little less laggy.

With the manual inputs, there is a limit to only 2 decimal places that we can input, but with equilibrium there are usually many decimal places down to very small numbers. I suggest that you find a way to allow for smaller decimals to be used as well as this can be impactful in generating good results.

All the existing features were implemented very well and I don't have any further improvements.

The simulations mathematically work, it's just that visually the animations sometimes bug out which is still very rare



Any suggestions for additional features to implement in the future?

4 responses

Similar as above just to implement more UI features and add more interactions. Also add a more in-depth description of the things that are happening within the simulation as well.

If you could add some visual representations of what occurs in LeChatelier's principle instead of just listing out the effects, that would be super cool as well.

Adding adjustable reaction conditions, such as temperature and pressure sliders, could help users explore how equilibrium shifts dynamically. A feature that provides step-by-step explanations of equilibrium changes based on user inputs would enhance understanding.

I would just work a little on the animation to make it visually show

Recommendations for Improvement - 136 words

- Add visual/interactive apps that can show the changes users modify in Lechatelier's principle, such as a vacuum with the ability to change pressure, and a visual to change the temperature.
- Improve manual inputs so that users can add more than two decimal places, since the majority of equilibrium reactions occur between small values between 0 and 1.
- Add adjustable reaction conditions and step-by step explanations of equilibrium changes, based on user inputs
- Add smoother animations and for the line chart, don't make the graph move along the x axis so the user can see the full change of the concentrations
- Patch bug fixes such as sometimes not having the particles combine into products

- Add more in depth descriptions of everything that occurs, explaining how all the operations and calculations are being processed.

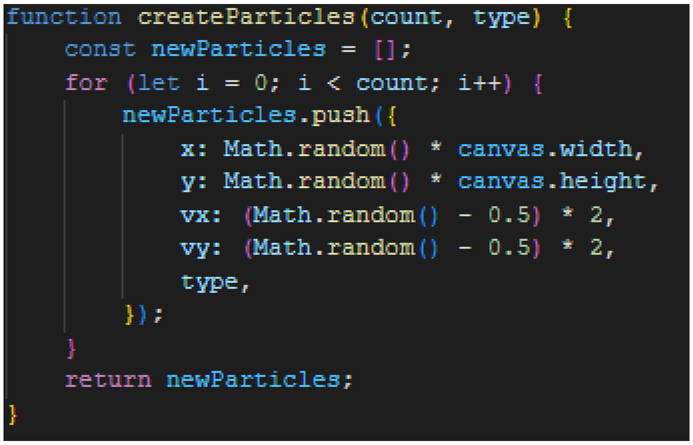
Appendix C1

Code functions for the particle simulator

3.1 Particle Functions

3.1.1 createParticles(count, type)

The purpose of this function is to generate a given number of particles of a specified type (e.g., "reactantA") with random positions and velocities inside the canvas.

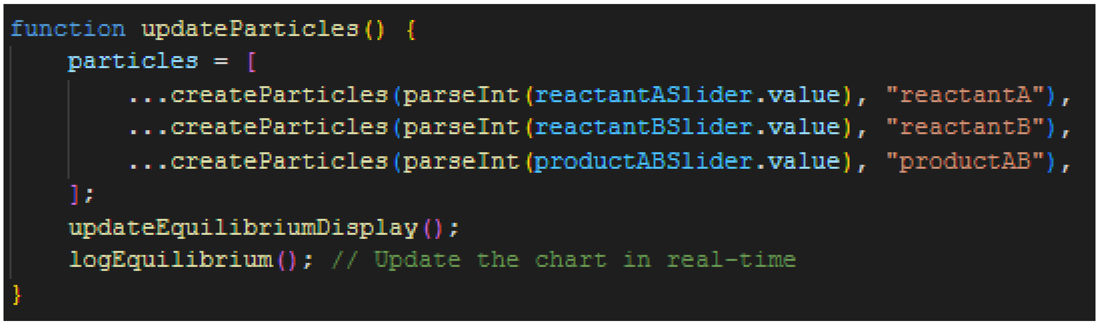


```
function createParticles(count, type) {
    const newParticles = [];
    for (let i = 0; i < count; i++) {
        newParticles.push({
            x: Math.random() * canvas.width,
            y: Math.random() * canvas.height,
            vx: (Math.random() - 0.5) * 2,
            vy: (Math.random() - 0.5) * 2,
            type,
        });
    }
    return newParticles;
}
```

The function first initializes an empty array newParticles, then uses loops count times to assign coordinates (x,y) within canvas bounds and add a velocity, and add a particle type

3.1.2 updateParticles()

The purpose of this function is to regenerate the simulation with the current values from the input sliders for each particle type.

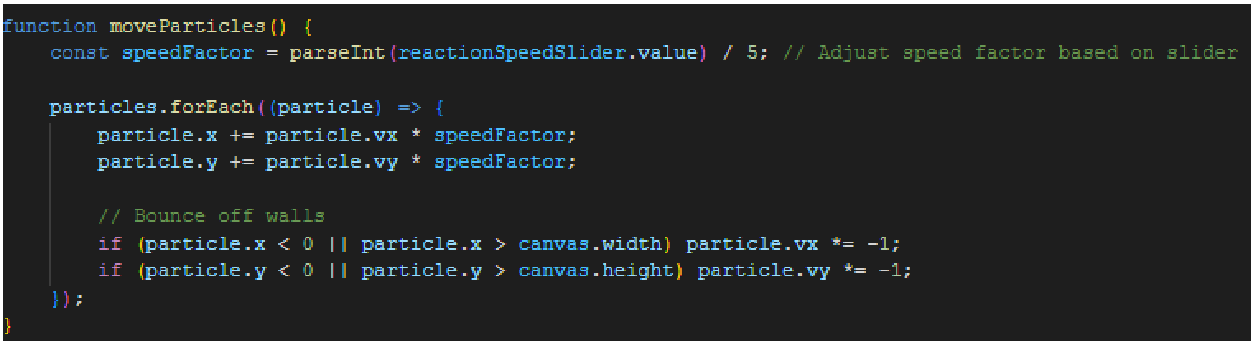


```
function updateParticles() {
    particles = [
        ...createParticles(parseInt(reactantASlider.value), "reactantA"),
        ...createParticles(parseInt(reactantBSlider.value), "reactantB"),
        ...createParticles(parseInt(productABSlider.value), "productAB"),
    ];
    updateEquilibriumDisplay();
    logEquilibrium(); // Update the chart in real-time
}
```

The function reads the current values from the sliders (Reactant A, Reactant B, Product AB), then calls the createParticles() function for the 3 respective types, and then combines them all back into one array to replace the global particles. Finally, it updates the current displayed equilibrium with updateEquilibriumDisplay() (3.7) and uses logEquilibrium() (3.8) to log the state for graphing.

3.1.3 moveParticles()

This function animates the particle movement in the canvas, giving a dynamic and fluid feel to the system.

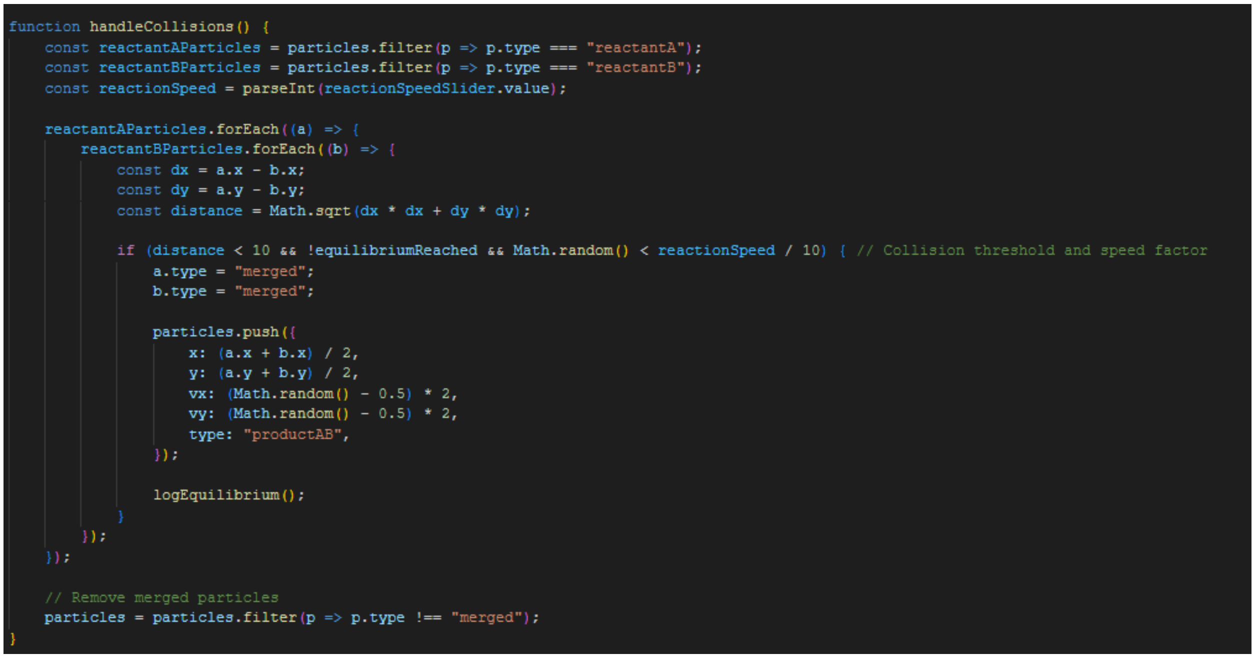


```
function moveParticles() {
    const speedFactor = parseInt(reactionSpeedSlider.value) / 5; // Adjust speed factor based on slider

    particles.forEach((particle) => {
        particle.x += particle.vx * speedFactor;
        particle.y += particle.vy * speedFactor;

        // Bounce off walls
        if (particle.x < 0 || particle.x > canvas.width) particle.vx *= -1;
        if (particle.y < 0 || particle.y > canvas.height) particle.vy *= -1;
    });
}
```

The function first parses the user input of reaction speed to determine the constant velocity scalar all particles are going to move by, then it iterates through the global particle array and updates its position based on the new velocity. Finally, the last two if statements implement bouncing off walls by reflecting the direction of the particle’s velocity.

3.1.4 handleCollisions()

This function detects collisions between A and B particles and, if conditions are met, create a new productAB particle to simulate a reaction.

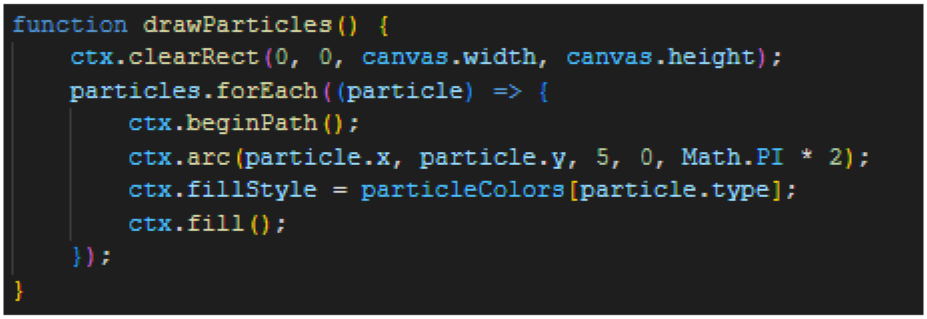


```
function handleCollisions() {
    const reactantAParticles = particles.filter(p => p.type === "reactantA");
    const reactantBParticles = particles.filter(p => p.type === "reactantB");
    const reactionSpeed = parseInt(reactionSpeedSlider.value);

    reactantAParticles.forEach((a) => {
        reactantBParticles.forEach((b) => {
            const dx = a.x - b.x;
            const dy = a.y - b.y;
            const distance = Math.sqrt(dx * dx + dy * dy);

            if (distance < 10 && !equilibriumReached && Math.random() < reactionSpeed / 10) { // Collision threshold and speed factor
                a.type = "merged";
                b.type = "merged";

                particles.push({
                    x: (a.x + b.x) / 2,
                    y: (a.y + b.y) / 2,
                    vx: (Math.random() - 0.5) * 2,
                    vy: (Math.random() - 0.5) * 2,
                    type: "productAB",
                });

                logEquilibrium();
            }
        });
    });

    // Remove merged particles
    particles = particles.filter(p => p.type !== "merged");
}
```

First, the function will filter the particle list to get arrays of just A and B types. Then, for each AB pair: calculate the distance between them, and if close enough, it will use a random check to give a probability of reaction occurring. If the reaction occurs, both of the reactants are marked to be removed by changing their type to "merged", and adds a new productAB at the collision point. After looping, the function will remove all "merged" particles from the system.

3.1.5 drawParticles()

This function simply renders the particles on the canvas based on their types and positions.

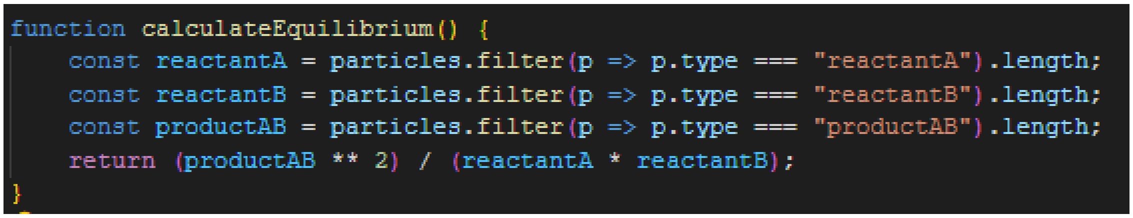


```
function drawParticles() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    particles.forEach((particle) => {
        ctx.beginPath();
        ctx.arc(particle.x, particle.y, 5, 0, Math.PI * 2);
        ctx.fillStyle = particleColors[particle.type];
        ctx.fill();
    });
}
```

The function clears the canvas at each frame to avoid overlapping visuals. Then, it iterates over every particle in the system and draws each as a circle of radius 5 pixels, and based on the type of particle (based on particle.type) colors the particles using predefined particleColors mapping.

3.2 calculateEquilibrium()

This function checks the equilibrium at every frame of the reaction. This is necessary to find at what point the particles need to stop reacting once equilibrium has been reached.



```
function calculateEquilibrium() {
    const reactantA = particles.filter(p => p.type === "reactantA").length;
    const reactantB = particles.filter(p => p.type === "reactantB").length;
    const productAB = particles.filter(p => p.type === "productAB").length;
    return (productAB ** 2) / (reactantA * reactantB);
}
```

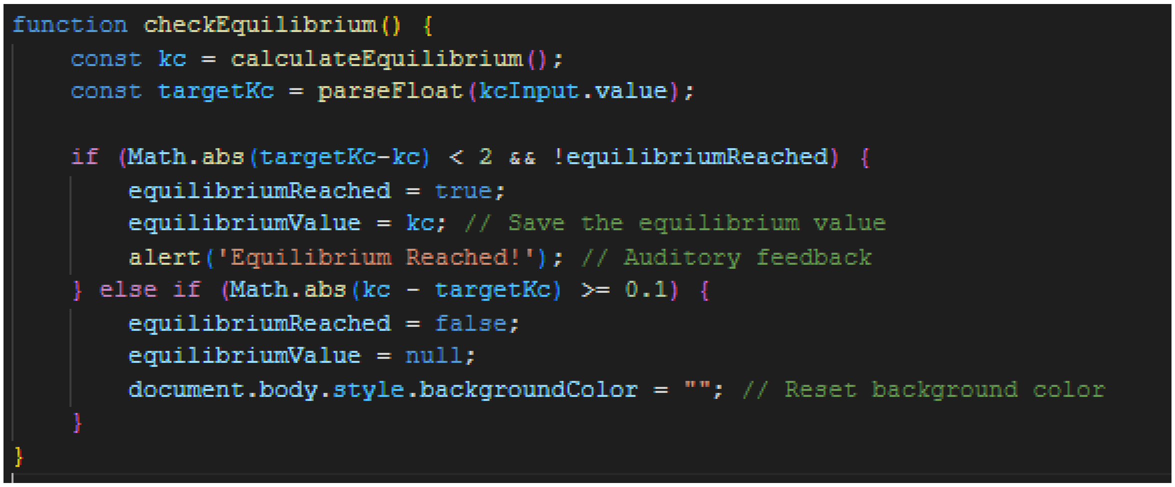
This function sorts through the particle array by each type, and finds the number of particles of each type still remaining on the canvas. Then, it uses the formula of products divided by the reactants

$$K = \frac{[AB]^2}{[A][B]}$$

to get the current value to compare to the equilibrium threshold.

3.3 checkEquilibrium()

This function continuously monitors and determines if the system has reached chemical equilibrium by comparing the current calculated K_c value to the user-defined equilibrium constant input.

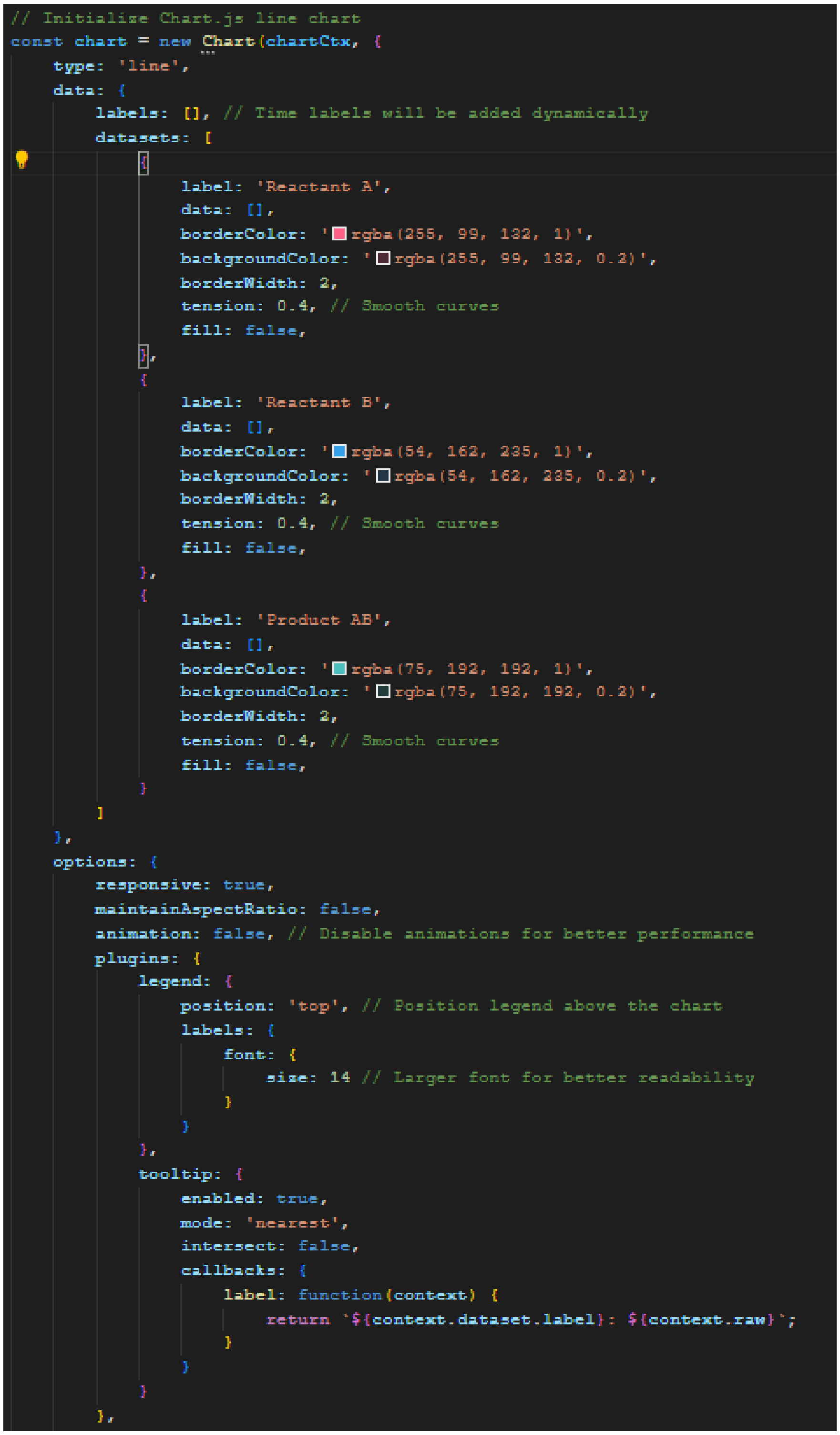


```
function checkEquilibrium() {
    const kc = calculateEquilibrium();
    const targetKc = parseFloat(kcInput.value);

    if (Math.abs(targetKc-kc) < 2 && !equilibriumReached) {
        equilibriumReached = true;
        equilibriumValue = kc; // Save the equilibrium value
        alert('Equilibrium Reached!'); // Auditory feedback
    } else if (Math.abs(kc - targetKc) >= 0.1) {
        equilibriumReached = false;
        equilibriumValue = null;
        document.body.style.backgroundColor = ""; // Reset background color
    }
}
```

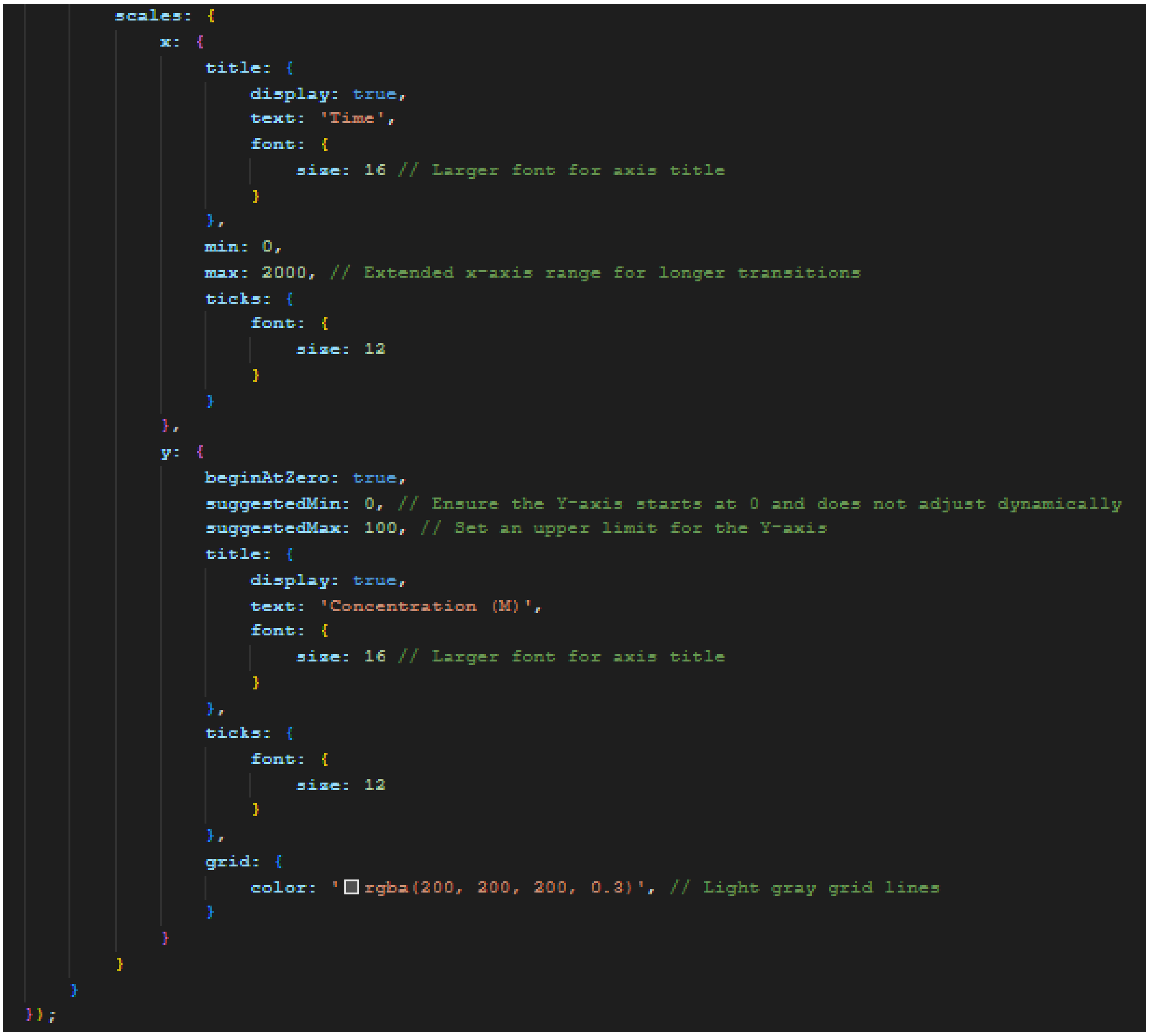
First, we call on the calculateEquilibrium (3.3) to check the current K_c value of the system. Then, we use an if statement to check if the current K_c is within 2 of the user's target K_c. If it is, then equilibriumReached boolean is set to true, and the equilibriumValue is saved as what it is currently at. Finally, a popup alert is displayed to let the user know that equilibrium has been reached. If the system is too far away from the equilibrium, everything will be reset back to the default, since the equilibrium cannot be reached.

3.4 Chart initialization

This code initializes a line chart using the Chart.js library.



```
// Initialize Chart.js line chart
const chart = new Chart(chartCtx, {
    type: 'line',
    data: {
        labels: [], // Time labels will be added dynamically
        datasets: [
            {
                label: 'Reactant A',
                data: [],
                borderColor: 'rgba(255, 99, 132, 1)',
                backgroundColor: 'rgba(255, 99, 132, 0.2)',
                borderWidth: 2,
                tension: 0.4, // Smooth curves
                fill: false,
            },
            {
                label: 'Reactant B',
                data: [],
                borderColor: 'rgba(54, 162, 235, 1)',
                backgroundColor: 'rgba(54, 162, 235, 0.2)',
                borderWidth: 2,
                tension: 0.4, // Smooth curves
                fill: false,
            },
            {
                label: 'Product AB',
                data: [],
                borderColor: 'rgba(75, 192, 192, 1)',
                backgroundColor: 'rgba(75, 192, 192, 0.2)',
                borderWidth: 2,
                tension: 0.4, // Smooth curves
                fill: false,
            }
        ]
    },
    options: {
        responsive: true,
        maintainAspectRatio: false,
        animation: false, // Disable animations for better performance
        plugins: {
            legend: {
                position: 'top', // Position legend above the chart
                labels: {
                    font: {
                        size: 14 // Larger font for better readability
                    }
                }
            },
            tooltip: {
                enabled: true,
                mode: 'nearest',
                intersect: false,
                callbacks: {
                    label: function(context) {
                        return `${context.dataset.label}: ${context.raw}`;
                    }
                }
            }
        },
```

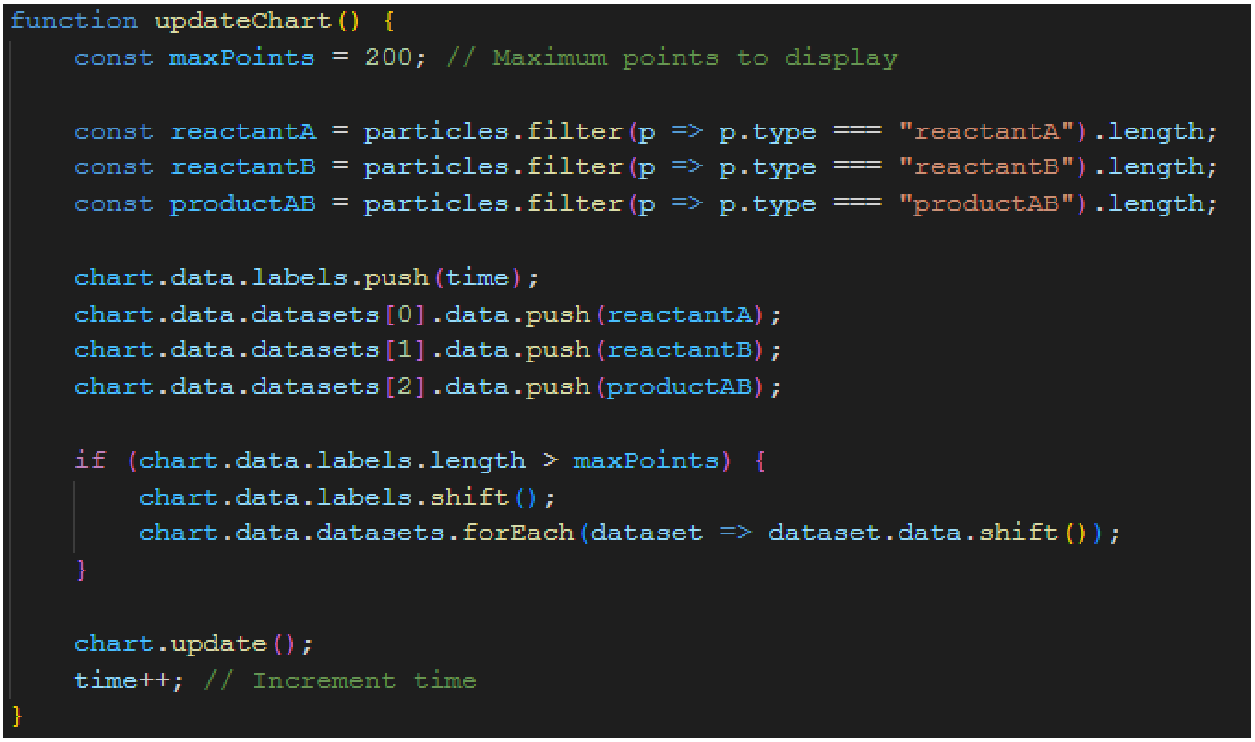


```
        scales: {
            x: {
                title: {
                    display: true,
                    text: 'Time',
                    font: {
                        size: 16 // Larger font for axis title
                    }
                },
                min: 0,
                max: 2000, // Extended x-axis range for longer transitions
                ticks: {
                    font: {
                        size: 12
                    }
                }
            },
            y: {
                beginAtZero: true,
                suggestedMin: 0, // Ensure the Y-axis starts at 0 and does not adjust dynamically
                suggestedMax: 100, // Set an upper limit for the Y-axis
                title: {
                    display: true,
                    text: 'Concentration (M)',
                    font: {
                        size: 16 // Larger font for axis title
                    }
                },
                ticks: {
                    font: {
                        size: 12
                    }
                },
                grid: {
                    color: '□rgba(200, 200, 200, 0.3)', // Light gray grid lines
                }
            }
        }
    }
});
```

This code initializes a dynamic line chart using the Chart.js library, which visually represents changes in concentration of chemical species over time. The chart is embedded within a canvas element (chartCanvas) and is designed to help simulate and observe chemical equilibrium behavior in real time. Specifically, it tracks the concentrations of Reactant A, Reactant B, and Product AB, allowing users to observe how they change throughout the simulation. It is initialized as a line chart, and sets X axis as time and Y axis as concentrations.

3.5 updateChart()

This function updates the chart at every second, plotting each point and creating a line graph for the user to visualize the change over time of the concentrations.

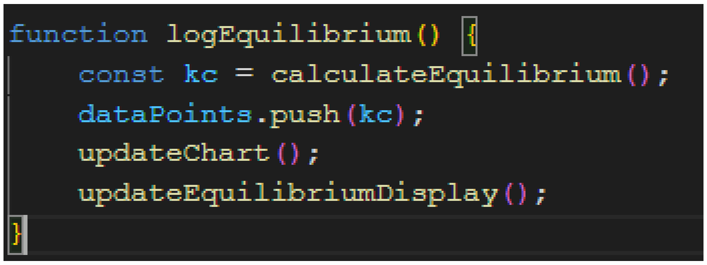


```
function updateChart() {
    const maxPoints = 200; // Maximum points to display

    const reactantA = particles.filter(p => p.type === "reactantA").length;
    const reactantB = particles.filter(p => p.type === "reactantB").length;
    const productAB = particles.filter(p => p.type === "productAB").length;

    chart.data.labels.push(time);
    chart.data.datasets[0].data.push(reactantA);
    chart.data.datasets[1].data.push(reactantB);
    chart.data.datasets[2].data.push(productAB);

    if (chart.data.labels.length > maxPoints) {
        chart.data.labels.shift();
        chart.data.datasets.forEach(dataset => dataset.data.shift());
    }

    chart.update();
    time++; // Increment time
}
```

This function dynamically updates a line chart to visualize the concentration of particles over time during the simulation. It counts the current number of particles of each type by filtering the global particles array, then appends these values to their corresponding datasets in the chart. The function also tracks time, adding each time step to the X-axis labels. To maintain performance and readability, it limits the chart to 200 data points, removing the oldest entries when necessary. After updating the data, it calls chart.update() to re-render the chart and increments the global time variable, ensuring the simulation progresses smoothly and reflects real-time changes in particle behavior and equilibrium.

3.6 logEquilibrium()

This function serves as the central logger during the simulation, capturing the current equilibrium state at each step.

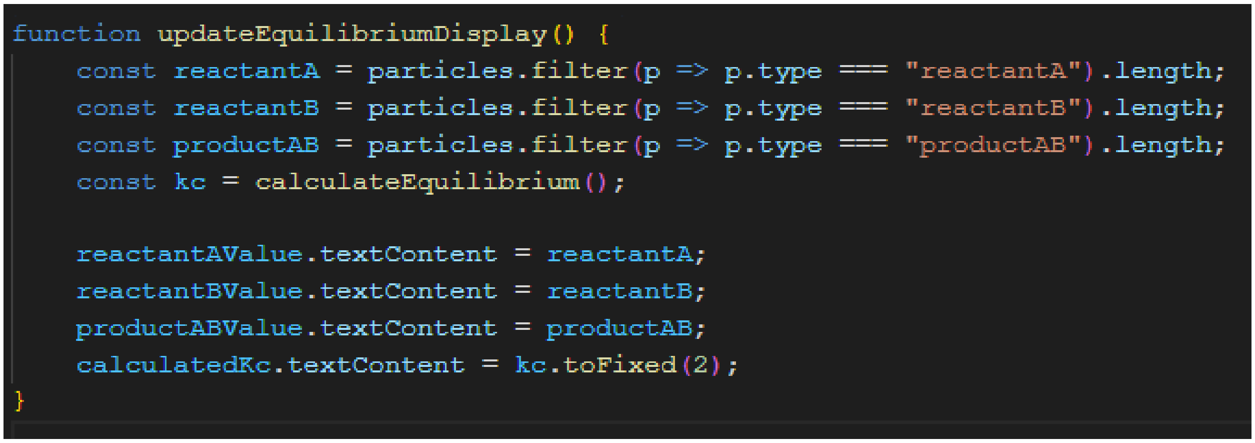


```
function logEquilibrium() {
    const kc = calculateEquilibrium();
    dataPoints.push(kc);
    updateChart();
    updateEquilibriumDisplay();
}
```

calculateEquilibrium() finds current K_c value, which is then stored in the dataPoints array, allowing a historical record of equilibrium values to be kept over time. After logging K_c, the function calls updateChart() to reflect the new concentrations on the line graph, and then updateEquilibriumDisplay() to update any on-screen numerical display of Kc.

3.7 updateEquilibriumDisplay()

This function updates the on-screen values that represent the current state of the reaction.

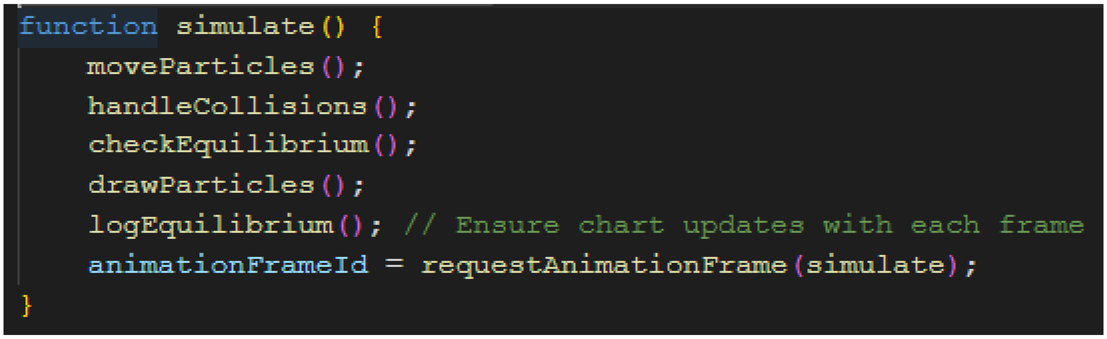


```
function updateEquilibriumDisplay() {
    const reactantA = particles.filter(p => p.type === "reactantA").length;
    const reactantB = particles.filter(p => p.type === "reactantB").length;
    const productAB = particles.filter(p => p.type === "productAB").length;
    const kc = calculateEquilibrium();

    reactantAValue.textContent = reactantA;
    reactantBValue.textContent = reactantB;
    productABValue.textContent = productAB;
    calculatedKc.textContent = kc.toFixed(2);
}
```

The function starts by filtering the global particles array to count how many particles exist for Reactant A, Reactant B, and Product AB. These values give a snapshot of the current concentrations in the simulation. It then calls calculateEquilibrium() to compute the equilibrium constant (Kc) based on these counts. Finally, it updates the corresponding HTML elements (reactantAValue, reactantBValue, productABValue, and calculatedKc) to reflect the current particle counts and the latest Kc value, formatted to two decimal places.

3.8 simulate()

This function drives the core animation loop of the particle simulation, updating the system frame by frame.

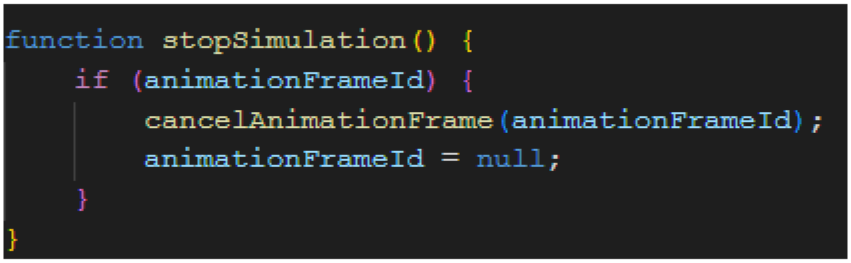


```
function simulate() {
    moveParticles();
    handleCollisions();
    checkEquilibrium();
    drawParticles();
    logEquilibrium(); // Ensure chart updates with each frame
    animationFrameId = requestAnimationFrame(simulate);
}
```

The function first calls moveParticles() to update each particle's position, simulating random motion. Next, handleCollisions() checks for and processes any reactions between particles based on proximity and reaction speed. After that, checkEquilibrium() evaluates whether the system has reached equilibrium by comparing the current Kc to the target. drawParticles() then renders the updated particle positions on the canvas, and logEquilibrium() records the current state, updating both the chart and equilibrium display. Finally, requestAnimationFrame(simulate) schedules the next frame, allowing the simulation to run continuously in real time.

3.9 stopSimulation()

This function halts the ongoing particle simulation by cancelling the scheduled animation frame.

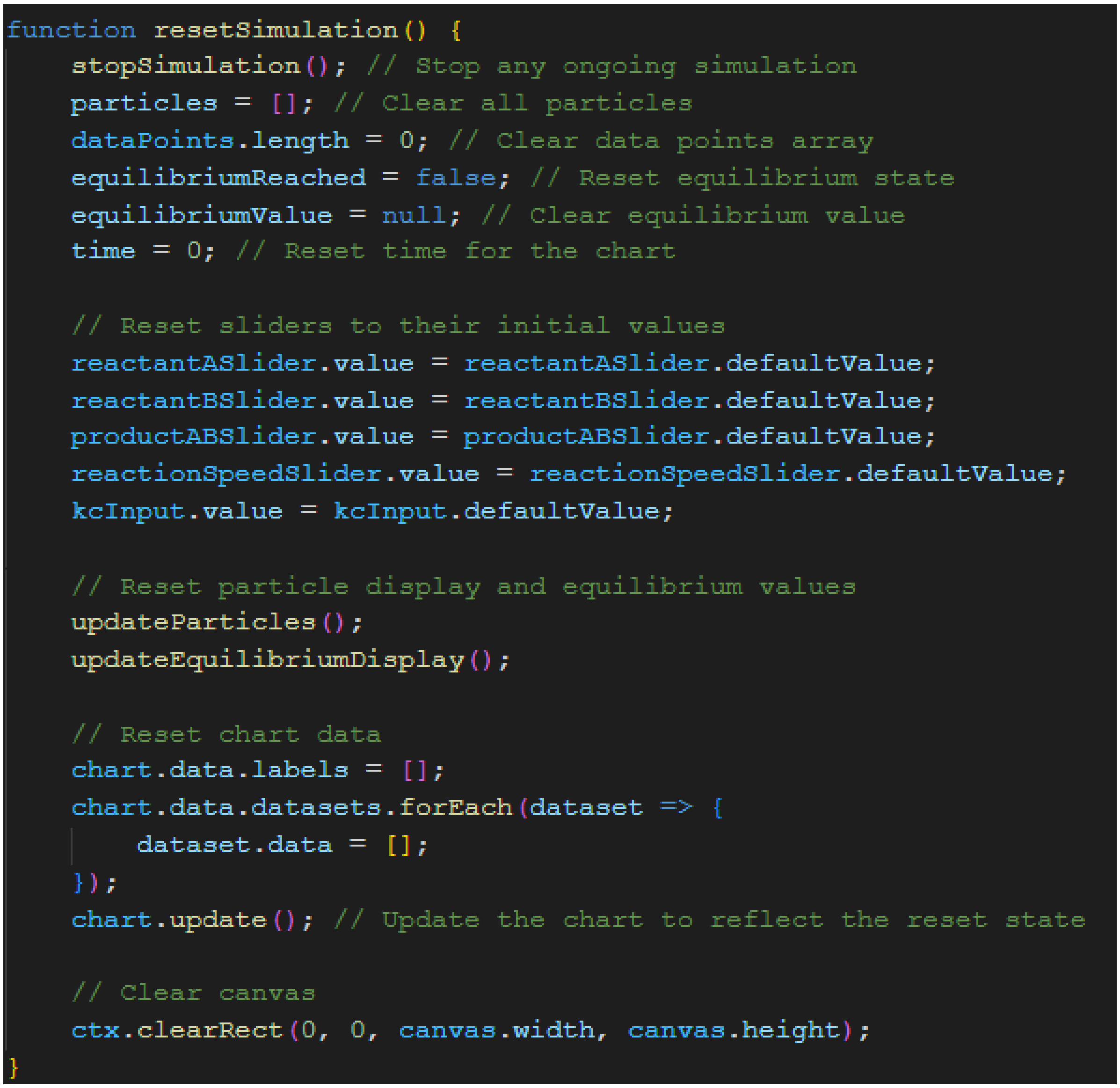


```
function stopSimulation() {
    if (animationFrameId) {
        cancelAnimationFrame(animationFrameId);
        animationFrameId = null;
    }
}
```

It checks if animationFrameId exists—indicating that the simulation loop is currently running—and then calls cancelAnimationFrame(animationFrameId) to stop it. After cancelling, it resets animationFrameId to null to ensure the simulation can be restarted cleanly later. This function effectively pauses the simulation without clearing any data or visuals, preserving the current state for review or resumption.

3.10 resetSimulation()

The resetSimulation() function completely re-initializes the particle simulation to its original state.

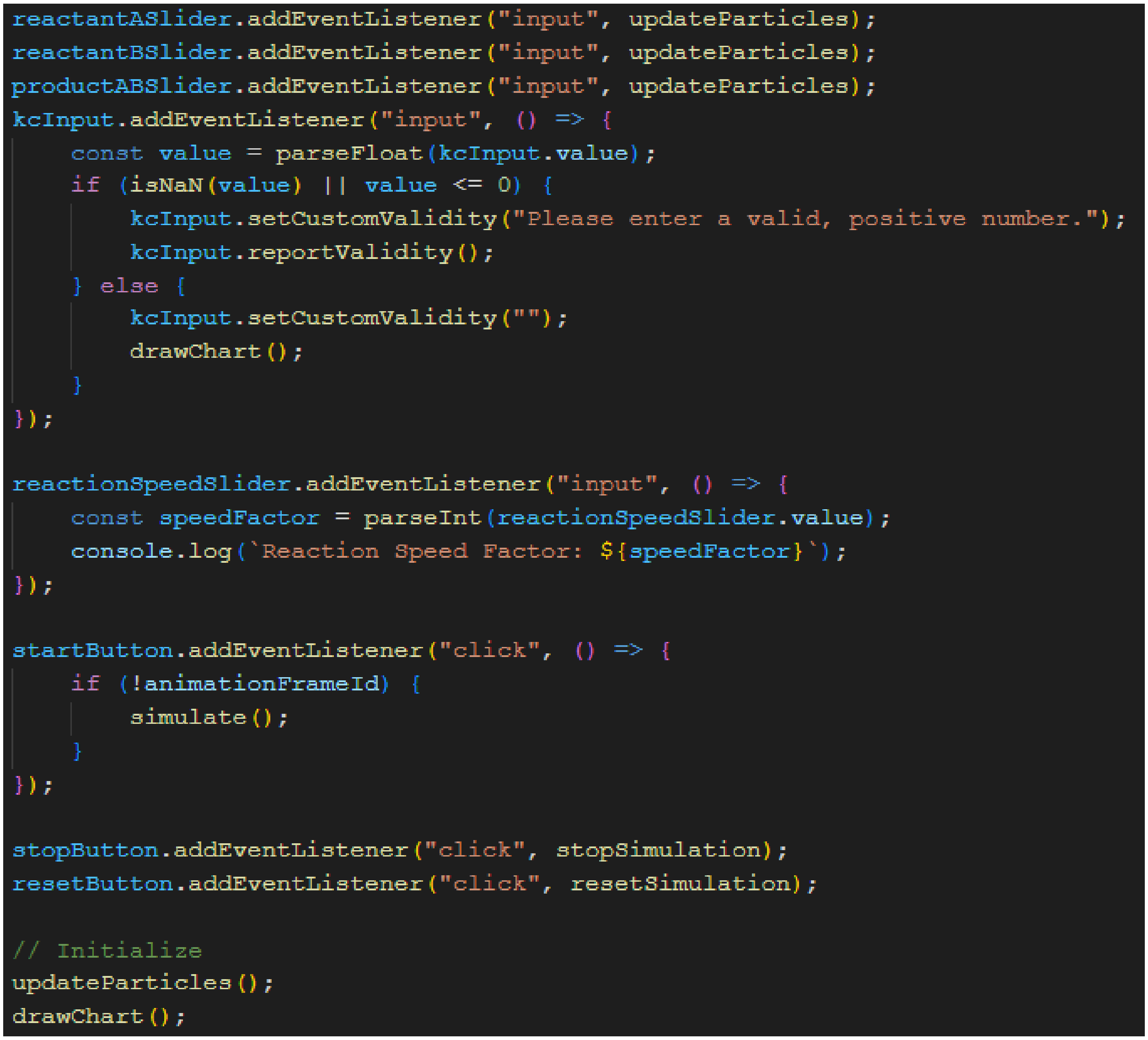


```
function resetSimulation() {
    stopSimulation(); // Stop any ongoing simulation
    particles = []; // Clear all particles
    dataPoints.length = 0; // Clear data points array
    equilibriumReached = false; // Reset equilibrium state
    equilibriumValue = null; // Clear equilibrium value
    time = 0; // Reset time for the chart

    // Reset sliders to their initial values
    reactantASlider.value = reactantASlider.defaultValue;
    reactantBSlider.value = reactantBSlider.defaultValue;
    productABSlider.value = productABSlider.defaultValue;
    reactionSpeedSlider.value = reactionSpeedSlider.defaultValue;
    kcInput.value = kcInput.defaultValue;

    // Reset particle display and equilibrium values
    updateParticles();
    updateEquilibriumDisplay();

    // Reset chart data
    chart.data.labels = [];
    chart.data.datasets.forEach(dataset => {
        dataset.data = [];
    });
    chart.update(); // Update the chart to reflect the reset state

    // Clear canvas
    ctx.clearRect(0, 0, canvas.width, canvas.height);
}
```

The function starts by calling stopSimulation() to ensure no animation is running. Then, it clears all particles and equilibrium data, resets flags like equilibriumReached an equilibriumValue, and sets the time counter back to zero. All input sliders (for reactants, products, reaction speed, and Kc) are restored to their default values to ensure a consistent starting point. The particle system and display values are updated via updateParticles() and updateEquilibriumDisplay(), and the chart's data is wiped clean before being refreshed with chart.update(). Finally, the canvas is cleared visually, removing any remaining particles from view.

3.1 Event Listeners and Initialization of chart/simulator

This code snippet handles the event listeners and initialization logic for the particle simulator's user interface.



```
reactantASlider.addEventListener("input", updateParticles);
reactantBSlider.addEventListener("input", updateParticles);
productABSlider.addEventListener("input", updateParticles);
kcInput.addEventListener("input", () => {
    const value = parseFloat(kcInput.value);
    if (isNaN(value) || value <= 0) {
        kcInput.setCustomValidity("Please enter a valid, positive number.");
        kcInput.reportValidity();
    } else {
        kcInput.setCustomValidity("");
        drawChart();
    }
});

reactionSpeedSlider.addEventListener("input", () => {
    const speedFactor = parseInt(reactionSpeedSlider.value);
    console.log(`Reaction Speed Factor: ${speedFactor}`);
});

startButton.addEventListener("click", () => {
    if (!animationFrameId) {
        simulate();
    }
});

stopButton.addEventListener("click", stopSimulation);
resetButton.addEventListener("click", resetSimulation);

// Initialize
updateParticles();
drawChart();
```

Event listeners are added so the app responds immediately to user changes of the input values. Buttons are added to stop and reset the simulation. Finally, updateParticles() and drawChart() are called once at the end to initialize the display with the default state when the page loads.