

Scientific Calculator

Hardware Project

EE1003 : Scientific Programming

GVV Sharma

Harshil Rathan Y EE24BTECH11064

Contents

1	Introduction	2
2	Components Required	2
3	Circuit Design	2
3.1	Pin Diagrams	2
3.2	Push Button Matrix	2
4	Working	4
5	Results	21

1 Introduction

This project implements a scientific calculator using an Arduino, an LCD display, and push buttons for input. The calculator supports basic arithmetic operations, matrix calculations, and differentiation. It processes user inputs through button presses, displays real-time results on the LCD, and utilizes avr-gcc for efficient computation and logic control.

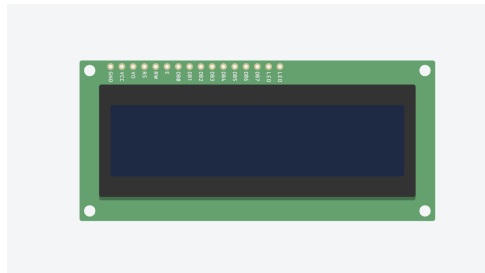
2 Components Required

1. Arduino UNO - 1
2. 16x2 LCD Display - 1
3. Push Buttons - 24
4. Breadboard - 2
5. Jumper cables
6. Potentiometer - 1

3 Circuit Design

3.1 Pin Diagrams

The pin diagram of LCD is given below

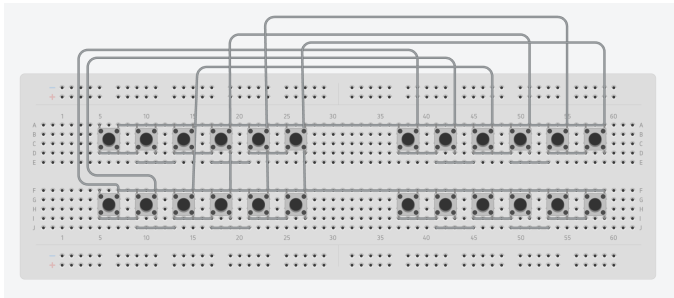


3.2 Push Button Matrix

- A push-button matrix is a method of reading multiple button inputs using a grid-like arrangement, reducing the number of required microcontroller pins. This is achieved by scanning rows and columns to detect button presses.
- Here, we used it to efficiently interface multiple push buttons with the Arduino, allowing user input for calculator operations while conserving I/O pins.
- Normally to implement 24 buttons we need 24 I/O pins, by using matrix connections we can reduce it to 10 pins

Connections

In my connections i have defined a 4x6 matrix for various functions



- Connect one pin of the push button to all the rows and the final element of the rows to the arduino pins
- Collect the pther pin of the push button to all the columns and the final element of each column to the arduino pins

LCD Connections

LCD Pin	Name	Arduino Pin	Description
1	VSS	GND	Ground
2	VDD	5V	Power Supply (5V)
3	V0	Potentiometer(MiddlePin)	Contrast Adjustment
4	RS	7	Register Select
5	RW	GND	Read/Write
6	E	6	Enable Pin
7	D0	-	Not used
8	D1	-	Not used
9	D2	-	Not used
10	D3	-	Not used
11	D4	5	Data Line 4
12	D5	4	Data Line 5
13	D6	3	Data Line 6
14	D7	2	Data Line 7
15	LED+	5V	LED Backlight Power
16	LED-	GND	LED Backlight Ground

Connect the other ends of the potentiometer to 5V and GND

Keypad Connections

Keypad Pin	Arduino Pin
Row 1	9
Row 2	8
Row 3	7
Row 4	6
Column 1	5
Column 2	4
Column 3	3
Column 4	2

4 Working

Power up the arduino and upload the following code

```

1  #include <avr/io.h>
2  #include <util/delay.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <math.h>
6  #include <ctype.h>
7  #include <stdio.h>
8  #include <stdint.h>
9
10 // LCD Pin Definitions
11 // LCD Pin Definitions
12 #define LCD_RS_DDR DDRD
13 #define LCD_RS_PORT PORTD
14 #define LCD_RS_PIN 7
15
16 #define LCD_E_DDR DDRD
17 #define LCD_E_PORT PORTD
18 #define LCD_E_PIN 6
19
20 #define LCD_D4_DDR DDRD
21 #define LCD_D4_PORT PORTD
22 #define LCD_D4_PIN 5
23
24 #define LCD_D5_DDR DDRD
25 #define LCD_D5_PORT PORTD
26 #define LCD_D5_PIN 4
27
28 #define LCD_D6_DDR DDRD
29 #define LCD_D6_PORT PORTD
30 #define LCD_D6_PIN 3
31
32 #define LCD_D7_DDR DDRD
33 #define LCD_D7_PORT PORTD
34 #define LCD_D7_PIN 2
35
36 // Keypad Row Pin Definitions (Moved from Port A to Port C)
37 #define ROW1_DDR DDRC
38 #define ROW1_PORT PORTC
39 #define ROW1_PIN PINC
40 #define ROW1_BIT 0

```

```

41
42 #define ROW2_DDR DDRC
43 #define ROW2_PORT PORTC
44 #define ROW2_PIN PINC
45 #define ROW2_BIT 1
46
47 #define ROW3_DDR DDRC
48 #define ROW3_PORT PORTC
49 #define ROW3_PIN PINC
50 #define ROW3_BIT 2
51
52 #define ROW4_DDR DDRC
53 #define ROW4_PORT PORTC
54 #define ROW4_PIN PINC
55 #define ROW4_BIT 3
56
57 // Keypad Column Pin Definitions (Unchanged, using Port B)
58 #define COL1_DDR DDRB
59 #define COL1_PORT PORTB
60 #define COL1_PIN PINB
61 #define COL1_BIT 0
62
63 #define COL2_DDR DDRB
64 #define COL2_PORT PORTB
65 #define COL2_PIN PINB
66 #define COL2_BIT 1
67
68 #define COL3_DDR DDRB
69 #define COL3_PORT PORTB
70 #define COL3_PIN PINB
71 #define COL3_BIT 2
72
73 #define COL4_DDR DDRB
74 #define COL4_PORT PORTB
75 #define COL4_PIN PINB
76 #define COL4_BIT 3
77
78 #define COL5_DDR DDRB
79 #define COL5_PORT PORTB
80 #define COL5_PIN PINB
81 #define COL5_BIT 4
82
83 #define COL6_DDR DDRB
84 #define COL6_PORT PORTB
85 #define COL6_PIN PINB
86 #define COL6_BIT 5
87
88
89 // Keypad dimensions
90 #define ROW_NUM 4
91 #define COLUMN_NUM 6
92
93 // Character mapping for keypad
94 char keys[ROW_NUM][COLUMN_NUM] = {
95     {'1', '2', '3', '+', '-', '*'},
96     {'4', '5', '6', '/', 'C', '='},
97     {'7', '8', '9', 'Q', 'M', 'D'},
98     {'0', 'e', 's', 'c', 't', 'S'}
99 };

```

```

100 // Arrays for row and column bits
101 uint8_t row_bits[ROW_NUM] = {ROW1_BIT, ROW2_BIT, ROW3_BIT, ROW4_BIT};
102 uint8_t col_bits[COLUMN_NUM] = {COL1_BIT, COL2_BIT, COL3_BIT, COL4_BIT, COL5_BIT,
103     COL6_BIT};
104
105 // Variables to store input and shift state
106 char expression[32] = "";
107 uint8_t expr_length = 0;
108 uint8_t lastCharOperator = 0; // Prevents consecutive operators
109 uint8_t shiftActive = 0; // Flag for shift function (Inverse trig functions)
110
111 // Matrix A and B inputs
112 float A[2][2], B[2][2], C[2][2];
113
114 // LCD Commands
115 #define LCD_CLEAR 0x01
116 #define LCD_HOME 0x02
117 #define LCD_ENTRY_MODE 0x06
118 #define LCD_DISPLAY_ON 0x0C
119 #define LCD_FUNCTION_SET 0x28
120 #define LCD_SET_DDRAM 0x80
121
122 char buffer[32]; // Buffer for string conversion
123
124 // Function to determine operator precedence
125 int precedence(char op) {
126     if (op == '+' || op == '-') return 1;
127     if (op == '*' || op == '/') return 2;
128     return 0;
129 }
130
131 // Function to apply an operation to two numbers
132 float applyOperation(float a, float b, char op) {
133     switch (op) {
134         case '+': return a + b;
135         case '-': return a - b;
136         case '*': return a * b;
137         case '/': return (b != 0) ? a / b : 0;
138     }
139     return 0;
140 }
141
142 // LCD Functions
143 void lcd_send_nibble(uint8_t nibble) {
144     // Set data pins according to nibble (higher 4 bits)
145     if (nibble & 0x01) LCD_D4_PORT |= (1 << LCD_D4_PIN); else LCD_D4_PORT &= ~(1 <<
        LCD_D4_PIN);
146     if (nibble & 0x02) LCD_D5_PORT |= (1 << LCD_D5_PIN); else LCD_D5_PORT &= ~(1 <<
        LCD_D5_PIN);
147     if (nibble & 0x04) LCD_D6_PORT |= (1 << LCD_D6_PIN); else LCD_D6_PORT &= ~(1 <<
        LCD_D6_PIN);
148     if (nibble & 0x08) LCD_D7_PORT |= (1 << LCD_D7_PIN); else LCD_D7_PORT &= ~(1 <<
        LCD_D7_PIN);
149
150     // Pulse E pin
151     LCD_E_PORT |= (1 << LCD_E_PIN);
152     _delay_us(1);
153     LCD_E_PORT &= ~(1 << LCD_E_PIN);

```

```

154     _delay_us(100);
155 }
156
157 void lcd_command(uint8_t cmd) {
158     LCD_RS_PORT &= ~(1 << LCD_RS_PIN); // RS = 0 for command
159
160     // Send high nibble
161     lcd_send_nibble(cmd >> 4);
162
163     // Send low nibble
164     lcd_send_nibble(cmd & 0x0F);
165
166     if (cmd == LCD_CLEAR || cmd == LCD_HOME)
167         _delay_ms(2); // These commands take longer
168     else
169         _delay_us(50);
170 }
171
172 void lcd_data(uint8_t data) {
173     LCD_RS_PORT |= (1 << LCD_RS_PIN); // RS = 1 for data
174
175     // Send high nibble
176     lcd_send_nibble(data >> 4);
177
178     // Send low nibble
179     lcd_send_nibble(data & 0x0F);
180
181     _delay_us(50);
182 }
183
184 void lcd_init(void) {
185     // Set data pins as output
186     LCD_RS_DDR |= (1 << LCD_RS_PIN);
187     LCD_E_DDR |= (1 << LCD_E_PIN);
188     LCD_D4_DDR |= (1 << LCD_D4_PIN);
189     LCD_D5_DDR |= (1 << LCD_D5_PIN);
190     LCD_D6_DDR |= (1 << LCD_D6_PIN);
191     LCD_D7_DDR |= (1 << LCD_D7_PIN);
192
193     _delay_ms(50); // Wait for LCD to power up
194
195     // Initialize in 4-bit mode
196     LCD_RS_PORT &= ~(1 << LCD_RS_PIN); // RS = 0 for command
197
198     // Send 0x03 three times (initialization sequence)
199     lcd_send_nibble(0x03);
200     _delay_ms(5);
201
202     lcd_send_nibble(0x03);
203     _delay_ms(5);
204
205     lcd_send_nibble(0x03);
206     _delay_ms(5);
207
208     // Now set to 4-bit mode
209     lcd_send_nibble(0x02);
210     _delay_ms(5);
211
212     // Now officially in 4-bit mode, send commands

```



```

213     lcd_command(LCD_FUNCTION_SET); // 4-bit, 2 lines, 5x8 font
214     lcd_command(LCD_DISPLAY_ON); // Display on, cursor off, no blink
215     lcd_command(LCD_CLEAR); // Clear display
216     lcd_command(LCD_ENTRY_MODE); // Increment cursor, no display shift
217
218     _delay_ms(2);
219 }
220
221 void lcd_string(const char *str) {
222     while (*str) {
223         lcd_data(*str++);
224     }
225 }
226
227 void lcd_set_cursor(uint8_t row, uint8_t col) {
228     uint8_t address;
229
230     if (row == 0)
231         address = 0x00 + col;
232     else if (row == 1)
233         address = 0x40 + col;
234     else if (row == 2)
235         address = 0x14 + col;
236     else
237         address = 0x54 + col;
238
239     lcd_command(LCD_SET_DDGRAM | address);
240 }
241
242 void lcd_clear(void) {
243     lcd_command(LCD_CLEAR);
244     _delay_ms(2);
245 }
246
247 // Keypad Functions
248 void keypad_init(void) {
249     // Set row pins as input with pull-ups
250     ROW1_DDR &= ~(1 << ROW1_BIT);
251     ROW2_DDR &= ~(1 << ROW2_BIT);
252     ROW3_DDR &= ~(1 << ROW3_BIT);
253     ROW4_DDR &= ~(1 << ROW4_BIT);
254
255     ROW1_PORT |= (1 << ROW1_BIT);
256     ROW2_PORT |= (1 << ROW2_BIT);
257     ROW3_PORT |= (1 << ROW3_BIT);
258     ROW4_PORT |= (1 << ROW4_BIT);
259
260     // Set column pins as output and set high
261     COL1_DDR |= (1 << COL1_BIT);
262     COL2_DDR |= (1 << COL2_BIT);
263     COL3_DDR |= (1 << COL3_BIT);
264     COL4_DDR |= (1 << COL4_BIT);
265     COL5_DDR |= (1 << COL5_BIT);
266     COL6_DDR |= (1 << COL6_BIT);
267
268     COL1_PORT |= (1 << COL1_BIT);
269     COL2_PORT |= (1 << COL2_BIT);
270     COL3_PORT |= (1 << COL3_BIT);
271     COL4_PORT |= (1 << COL4_BIT);

```

```

272 COL5_PORT |= (1 << COL5_BIT);
273 COL6_PORT |= (1 << COL6_BIT);
274 }
275
276 char get_key(void) {
277     uint8_t r, c;
278
279     // Check for keypress by scanning the keypad
280     for (c = 0; c < COLUMN_NUM; c++) {
281         // Set current column to low
282         switch(c) {
283             case 0: COL1_PORT &= ~(1 << COL1_BIT); break;
284             case 1: COL2_PORT &= ~(1 << COL2_BIT); break;
285             case 2: COL3_PORT &= ~(1 << COL3_BIT); break;
286             case 3: COL4_PORT &= ~(1 << COL4_BIT); break;
287             case 4: COL5_PORT &= ~(1 << COL5_BIT); break;
288             case 5: COL6_PORT &= ~(1 << COL6_BIT); break;
289         }
290
291         _delay_us(10); // Small delay for stabilization
292
293         // Check each row
294         for (r = 0; r < ROW_NUM; r++) {
295             uint8_t row_val = 0;
296
297             switch(r) {
298                 case 0: row_val = !(ROW1_PIN & (1 << ROW1_BIT)); break;
299                 case 1: row_val = !(ROW2_PIN & (1 << ROW2_BIT)); break;
300                 case 2: row_val = !(ROW3_PIN & (1 << ROW3_BIT)); break;
301                 case 3: row_val = !(ROW4_PIN & (1 << ROW4_BIT)); break;
302             }
303
304             if (row_val) { // Key is pressed
305                 // Set column back to high
306                 switch(c) {
307                     case 0: COL1_PORT |= (1 << COL1_BIT); break;
308                     case 1: COL2_PORT |= (1 << COL2_BIT); break;
309                     case 2: COL3_PORT |= (1 << COL3_BIT); break;
310                     case 3: COL4_PORT |= (1 << COL4_BIT); break;
311                     case 4: COL5_PORT |= (1 << COL5_BIT); break;
312                     case 5: COL6_PORT |= (1 << COL6_BIT); break;
313                 }
314
315                 _delay_ms(20); // Debounce delay
316                 return keys[r][c];
317             }
318         }
319
320         // Set column back to high
321         switch(c) {
322             case 0: COL1_PORT |= (1 << COL1_BIT); break;
323             case 1: COL2_PORT |= (1 << COL2_BIT); break;
324             case 2: COL3_PORT |= (1 << COL3_BIT); break;
325             case 3: COL4_PORT |= (1 << COL4_BIT); break;
326             case 4: COL5_PORT |= (1 << COL5_BIT); break;
327             case 5: COL6_PORT |= (1 << COL6_BIT); break;
328         }
329     }
330 }

```

```

331     return 0; // No key pressed
332 }
333
334 // Function to evaluate an expression using BODMAS
335 float evaluateExpression(char *expr) {
336     float values[16]; // Stack for numbers
337     char ops[16]; // Stack for operators
338     int valTop = -1, opTop = -1;
339     int i = 0;
340
341     while (expr[i] != '\0') {
342         if (isdigit(expr[i]) || expr[i] == '.') {
343             char num[16] = "";
344             int j = 0;
345             while (expr[i] != '\0' && (isdigit(expr[i]) || expr[i] == '.')) {
346                 num[j++] = expr[i++];
347             }
348             num[j] = '\0';
349             values[++valTop] = atof(num);
350             i--; // Adjust index
351         } else {
352             while (opTop >= 0 && precedence(ops[opTop]) >= precedence(expr[i])) {
353                 float b = values[valTop--];
354                 float a = values[valTop--];
355                 char op = ops[opTop--];
356                 values[++valTop] = applyOperation(a, b, op);
357             }
358             ops[++opTop] = expr[i];
359         }
360         i++;
361     }
362
363     while (opTop >= 0) {
364         float b = values[valTop--];
365         float a = values[valTop--];
366         char op = ops[opTop--];
367         values[++valTop] = applyOperation(a, b, op);
368     }
369
370     return values[0]; // Final result
371 }
372
373 // Function to handle trigonometric calculations (sin, cos, tan)
374 float trigFunction(char func, float angle) {
375     // Convert angle from degrees to radians before performing trig operations
376     float radians = angle * (M_PI / 180.0);
377
378     if (shiftActive) { // Inverse trigonometric functions (asin, acos, atan)
379         switch (func) {
380             case 's': // Inverse sin (asin)
381                 if (angle >= -1 && angle <= 1) {
382                     return asin(angle) * (180.0 / M_PI); // Convert result back to degrees
383                 } else {
384                     lcd_clear();
385                     lcd_string("asin: Invalid input");
386                     _delay_ms(2000);
387                     return 0; // Error message if out of range
388                 }
389             case 'c': // Inverse cos (acos)

```

```

390     if (angle >= -1 && angle <= 1) {
391         return acos(angle) * (180.0 / M_PI);
392     } else {
393         lcd_clear();
394         lcd_string("acos: Invalid input");
395         _delay_ms(2000);
396         return 0; // Error message if out of range
397     }
398     case 't': // Inverse tan (atan)
399         return atan(angle) * (180.0 / M_PI); // atan is valid for any real number
400 }
401 } else { // Regular sin, cos, tan functions
402     switch (func) {
403         case 's': // sin
404             return sin(radians);
405         case 'c': // cos
406             return cos(radians);
407         case 't': // tan
408             return tan(radians);
409     }
410 }
411 return 0;
412 }
413
414 // Function to handle the 'e^x' functionality
415 void handleExponent(void) {
416     lcd_clear();
417     lcd_string("Enter x:");
418     char numStr[16] = "";
419     int numIdx = 0;
420
421     while (1) {
422         char key = get_key();
423         if (key) {
424             if (isdigit(key) || key == '.' || key == '-') {
425                 numStr[numIdx++] = key;
426                 numStr[numIdx] = '\0';
427                 lcd_data(key);
428             }
429             else if (key == '=') {
430                 float exponent = atof(numStr);
431                 float result = exp(exponent); // Calculate e^x
432                 lcd_clear();
433                 lcd_string("e^x = ");
434
435                 // Convert float to string
436                 dtostrf(result, 6, 2, buffer);
437                 lcd_string(buffer);
438
439                 _delay_ms(2000);
440                 lcd_clear();
441                 break;
442             }
443         }
444         _delay_ms(100); // Debounce
445     }
446 }
447
448 // Function to input matrix A and B

```

```

449 float getMatrixElement(char matrixName, int row, int col) {
450     lcd_clear();
451     lcd_string("Enter_");
452     lcd_data(matrixName);
453     lcd_string("[");
454     lcd_data('0' + row);
455     lcd_string("][");
456     lcd_data('0' + col);
457     lcd_string("]:_");
458
459     char inputStr[16] = "";
460     int inputIdx = 0;
461
462     while (1) {
463         char key = get_key();
464
465         if (key) {
466             if ((key >= '0' && key <= '9') || key == '.' || key == '-') { // Allow
467                 digits, decimal, and negative sign
468                 if (inputIdx < 15) { // Prevent buffer overflow
469                     inputStr[inputIdx++] = key;
470                     inputStr[inputIdx] = '\0';
471                     lcd_data(key); // Display typed key
472                 }
473             } else if (key == '=') { // Confirm input
474                 break;
475             }
476             _delay_ms(100); // Debounce delay
477         }
478     }
479
480     return atof(inputStr); // Convert input string to float
481 }
482
483 void inputMatrices(void) {
484     lcd_clear();
485     lcd_string("Enter_A:");
486
487     A[0][0] = getMatrixElement('A', 1, 1); // A11
488     _delay_ms(500);
489
490     A[0][1] = getMatrixElement('A', 1, 2); // A12
491     _delay_ms(500);
492
493     A[1][0] = getMatrixElement('A', 2, 1); // A21
494     _delay_ms(500);
495
496     A[1][1] = getMatrixElement('A', 2, 2); // A22
497     _delay_ms(500);
498
499     lcd_clear();
500     lcd_string("Enter_B:");
501
502     B[0][0] = getMatrixElement('B', 1, 1); // B11
503     _delay_ms(500);
504
505     B[0][1] = getMatrixElement('B', 1, 2); // B12
506     _delay_ms(500);

```

```

507 B[1][0] = getMatrixElement('B', 2, 1); // B21
508 _delay_ms(500);
509
510
511 B[1][1] = getMatrixElement('B', 2, 2); // B22
512 _delay_ms(500);
513 }
514
515 // Function to perform matrix addition
516
517 void matrixAddition(void) {
518     for (int i = 0; i < 2; i++) {
519         for (int j = 0; j < 2; j++) {
520             C[i][j] = A[i][j] + B[i][j]; // Correct formula for addition
521         }
522     }
523 }
524
525 // Function to perform matrix multiplication
526 void matrixMultiplication(void) {
527     float tempC[2][2] = {0}; // Temporary matrix to avoid overwriting errors
528
529     for (int i = 0; i < 2; i++) {
530         for (int j = 0; j < 2; j++) {
531             tempC[i][j] = (A[i][0] * B[0][j]) + (A[i][1] * B[1][j]); // Correct
                    multiplication formula
532         }
533     }
534
535     // Copy result to C
536     for (int i = 0; i < 2; i++) {
537         for (int j = 0; j < 2; j++) {
538             C[i][j] = tempC[i][j];
539         }
540     }
541 }
542
543
544
545 // Function to handle shift toggle
546 void toggleShift(void) {
547     shiftActive = !shiftActive;
548     lcd_clear();
549     if (shiftActive) {
550         lcd_string("Inverse_Mode");
551     } else {
552         lcd_string("Calculator_Ready");
553         _delay_ms(2000);
554     }
555     _delay_ms(1000); // Show shift status for 1 second
556     lcd_clear();
557 }
558
559 // Function to get input for angle
560 float getInput(void) {
561     lcd_clear();
562     lcd_string("Enter_Angle:_");
563     char angleStr[16] = "";
564     int angleIdx = 0;

```

```

565 while (1) {
566     char key = get_key();
567     if (key) {
568         if (isdigit(key) || key == '.') {
569             angleStr[angleIdx++] = key;
570             angleStr[angleIdx] = '\0';
571             lcd_data(key);
572         } else if (key == '=') {
573             break;
574         }
575     }
576 }
577 _delay_ms(100); // Debounce
578 }
579
580 return atof(angleStr); // Convert input to float and return
581 }
582
583 // Function to get quadratic input
584 char buffer1[16], buffer2[16];
585
586 float getQuadraticInput(const char* prompt) {
587     lcd_clear();
588     lcd_string(prompt);
589     char inputStr[16] = ""; // Buffer for input
590     int inputIdx = 0;
591
592     while (1) {
593         char key = get_key(); // Read keypress
594         if (key) {
595             if (isdigit(key) || key == '.' || key == '-') { // Support negative numbers
596                 if (inputIdx < 15) { // Prevent buffer overflow
597                     inputStr[inputIdx++] = key;
598                     inputStr[inputIdx] = '\0';
599                     lcd_data(key);
600                 }
601             } else if (key == '=') { // Enter key (confirmation)
602                 break;
603             }
604         }
605         _delay_ms(100); // Debounce
606     }
607
608     return atof(inputStr); // Convert input string to float
609 }
610
611 // Function to get differentiation input
612 void getDifferentiationInput(float coeffs[], float powers[], int *termCount) {
613     lcd_clear();
614     lcd_string("Enter f(x):");
615     _delay_ms(1000);
616
617     char numStr[16] = "";
618     int numIdx = 0;
619     *termCount = 0;
620     uint8_t expectingPower = 0;
621     uint8_t expectingOperator = 0;
622
623

```

```

624 while (1) {
625     char key = get_key();
626     if (key) {
627         if (isdigit(key) || key == '.') {
628             numStr[numIdx++] = key;
629             numStr[numIdx] = '\0';
630             lcd_data(key);
631             expectingOperator = 0;
632         }
633         else if (key == 'x' || key == '*') { // Treat '*' as 'x' (multiplication)
634             if (numIdx > 0) { // If there is a coefficient before 'x'
635                 coeffs[*termCount] = atof(numStr);
636             } else {
637                 coeffs[*termCount] = 1; // Default coefficient is 1 if not specified
638             }
639             numIdx = 0;
640             numStr[0] = '\0';
641             lcd_data('x'); // Show 'x' for multiplication
642             expectingPower = 1; // Expecting the power after 'x'
643         }
644         else if (key == '+' || key == '-') { // Handle + and - for next term
645             if (numIdx > 0 && expectingPower) {
646                 powers[*termCount] = atof(numStr);
647                 numIdx = 0;
648                 numStr[0] = '\0';
649             } else if (numIdx > 0) {
650                 coeffs[*termCount] = atof(numStr);
651                 powers[*termCount] = 0; // If no 'x', power is 0 (constant term)
652                 numIdx = 0;
653                 numStr[0] = '\0';
654             }
655
656             if (expectingPower && numIdx == 0) {
657                 powers[*termCount] = 1; // Default power is 1 if not specified after 'x'
658             }
659
660             (*termCount)++;
661             lcd_data(key);
662             expectingPower = 0;
663             expectingOperator = 1;
664         }
665         else if (key == '=') { // Finish input
666             if (numIdx > 0) {
667                 if (expectingPower) {
668                     powers[*termCount] = atof(numStr);
669                 } else {
670                     coeffs[*termCount] = atof(numStr);
671                     powers[*termCount] = 0; // Constant term
672                 }
673             }
674
675             if (expectingPower && numIdx == 0) {
676                 powers[*termCount] = 1; // Default power is 1 if not specified after 'x'
677             }
678
679             (*termCount)++;
680             break;
681         }
682     }

```



```

683     _delay_ms(100); // Debounce
684 }
685 }
686
687 // Differentiation function
688 void differentiateFunction(void) {
689     float coeffs[10] = {0}, powers[10] = {0};
690     int termCount = 0;
691
692     getDifferentiationInput(coeffs, powers, &termCount);
693
694     lcd_clear();
695     lcd_string("f(x)=");
696     uint8_t firstTerm = 1;
697     for (int i = 0; i < termCount; i++) {
698         if (coeffs[i] != 0) { // Only print non-zero coefficients
699             if (!firstTerm && coeffs[i] > 0) lcd_data('+');
700
701             // Convert coefficient to string
702             dtostrf(coeffs[i], 3, 1, buffer);
703             lcd_string(buffer);
704
705             if (powers[i] != 0) { // If not a constant term
706                 lcd_data('x');
707
708                 if (powers[i] != 1) { // Only show power if not 1
709                     lcd_data('^');
710
711                     // Convert power to string
712                     dtostrf(powers[i], 2, 0, buffer);
713                     lcd_string(buffer);
714                 }
715             }
716
717             firstTerm = 0;
718         }
719     }
720
721     lcd_set_cursor(0, 1);
722     lcd_string("f'(x)=");
723     firstTerm = 1;
724     for (int i = 0; i < termCount; i++) {
725         if (powers[i] != 0 && coeffs[i] != 0) {
726             float derivativeCoeff = coeffs[i] * powers[i];
727             float derivativePower = powers[i] - 1;
728
729             if (!firstTerm && derivativeCoeff > 0) lcd_data('+');
730
731             // Convert derivative coefficient to string
732             dtostrf(derivativeCoeff, 3, 1, buffer);
733             lcd_string(buffer);
734
735             if (derivativePower != 0) { // Only show x if power is not 0 after
              differentiation
736                 lcd_data('x');
737
738                 if (derivativePower != 1) { // Only show power if not 1
739                     lcd_data('^');
740

```

```

741     // Convert derivative power to string
742     dtostrf(derivativePower, 2, 0, buffer);
743     lcd_string(buffer);
744 }
745 }
746
747     firstTerm = 0;
748 }
749 }
750
751 _delay_ms(5000);
752 lcd_clear();
753 }
754
755 // Function to solve quadratic equations
756 void inputQuadraticCoefficients(void) {
757     float a2, a1, a0;
758
759     // Prompt and get inputs
760     a2 = getQuadraticInput("Enter a2:");
761     _delay_ms(500); // Short delay before next prompt
762
763     a1 = getQuadraticInput("Enter a1:");
764     _delay_ms(500);
765
766     a0 = getQuadraticInput("Enter a0:");
767     _delay_ms(500);
768
769     // Check if it's a valid quadratic equation
770     if (a2 == 0) {
771         lcd_clear();
772         lcd_string("Not a quadratic");
773         _delay_ms(2000);
774         return;
775     }
776
777     // Calculate discriminant
778     float delta = a1 * a1 - 4 * a0 * a2;
779     lcd_clear();
780
781     if (delta < 0) {
782         lcd_string("No real roots");
783     } else {
784         float root1 = (-a1 + sqrt(delta)) / (2 * a2);
785         float root2 = (-a1 - sqrt(delta)) / (2 * a2);
786
787         // Convert roots to strings
788         dtostrf(root1, 6, 2, buffer1);
789         dtostrf(root2, 6, 2, buffer2);
790
791         // **FIX: Clear LCD again before printing**
792         lcd_clear();
793         _delay_ms(50); // Small delay for LCD stability
794
795         // **Ensure correct LCD positioning**
796         lcd_set_cursor(0, 0);
797         lcd_string("Root1:");
798         lcd_string(buffer1);
799

```

```

800     lcd_set_cursor(0, 1);
801     lcd_string("Root2:");
802     lcd_string(buffer2);
803 }
804 _delay_ms(3000);
805 }
806 // Function to display the result matrix C
807 void displayMatrix(void) {
808     lcd_clear();
809
810     char buffer[10];
811
812     // Display first row of result matrix C (C11 C12)
813     lcd_set_cursor(0, 0); // First row
814     dtostrf(C[0][0], 6, 2, buffer);
815     lcd_string(buffer);
816     lcd_string(" ");
817     dtostrf(C[0][1], 6, 2, buffer);
818     lcd_string(buffer);
819
820     // **FORCE CURSOR TO NEXT LINE**
821     _delay_ms(3000); // Small delay to allow LCD to process
822     lcd_set_cursor(0, 1); // Move to second row
823
824     // Display second row of result matrix C (C21 C22)
825     dtostrf(C[1][0], 6, 2, buffer);
826     lcd_string(buffer);
827     lcd_string(" ");
828     dtostrf(C[1][1], 6, 2, buffer);
829     lcd_string(buffer);
830 }
831
832 void matrixOperations(void) {
833     lcd_clear();
834     lcd_string("Matrix_Ops");
835     _delay_ms(500);
836
837     // Input Matrices
838     inputMatrices();
839
840     // Ask user for operation choice
841     lcd_clear();
842     lcd_string("3: A+B 4: A*B");
843
844     char key;
845     while (1) {
846         key = get_key();
847         if (key == '3') {
848             matrixAddition();
849             displayMatrix();
850             break;
851         } else if (key == '4') {
852             matrixMultiplication();
853             displayMatrix();
854             break;
855         }
856     }
857 }
858 }

```

```

859 int main(void) {
860     // Initialize LCD and Keypad
861     lcd_init();
862     keypad_init();
863
864     // Display welcome message
865     lcd_string("Sci_Calculator");
866     _delay_ms(2000);
867     lcd_clear();
868
869     while (1) {
870         char key = get_key();
871
872         if (key) {
873             if (key == 'C') { // C button clears the screen
874                 lcd_clear(); // Clears the LCD
875                 expr_length = 0;
876                 expression[0] = '\0';
877                 lastCharOperator = 0;
878             }
879             else if (key == 'S') { // Shift button toggles inverse mode
880                 toggleShift();
881             }
882             else if (key == 'e') { // Exponential function (e^x)
883                 handleExponent();
884             }
885             else if (key == 'Q') { // Quadratic equation solver
886                 inputQuadraticCoefficients();
887             }
888             else if (key == 'D') { // Differentiation
889                 differentiateFunction();
890             }
891             else if (key == 'M') { // Matrix operations
892                 lcd_clear();
893                 matrixOperations();
894             }
895
896             else if (key == 's' || key == 'c' || key == 't') { // Trigonometric functions
897                 float angle = getInput();
898                 float result = trigFunction(key, angle);
899
900                 lcd_clear();
901                 if (shiftActive) {
902                     lcd_data('a'); // Display 'a' for inverse trig functions
903                 }
904
905                 if (key == 's') lcd_string("sin(");
906                 else if (key == 'c') lcd_string("cos(");
907                 else if (key == 't') lcd_string("tan(");
908
909                 // Convert angle to string
910                 dtostrf(angle, 4, 1, buffer);
911                 lcd_string(buffer);
912                 lcd_string(")=");
913
914                 // Convert result to string
915                 dtostrf(result, 6, 4, buffer);
916                 lcd_string(buffer);
917

```

```

918     _delay_ms(3000);
919     lcd_clear();
920 }
921 else if (key == '=') { // Evaluate expression
922     if (expr_length > 0) {
923         float result = evaluateExpression(expression);
924         lcd_clear();
925         lcd_string("=");
926
927         // Convert result to string
928         dtostrf(result, 10, 4, buffer);
929         lcd_string(buffer);
930
931         _delay_ms(3000);
932         lcd_clear();
933
934         // Reset expression for a new calculation
935         expr_length = 0;
936         expression[0] = '\0';
937         lastCharOperator = 0;
938     }
939 }
940 else { // Regular input (digits, operators)
941     if (isdigit(key) || key == '.' || key == '-') {
942         if (expr_length < 31) { // Check for buffer overflow
943             expression[expr_length++] = key;
944             expression[expr_length] = '\0';
945             lcd_data(key);
946             lastCharOperator = 0;
947         }
948     }
949     else if (key == '+' || key == '-' || key == '*' || key == '/') {
950         if (expr_length > 0 && !lastCharOperator) { // Prevent consecutive operators
951             if (expr_length < 31) {
952                 expression[expr_length++] = key;
953                 expression[expr_length] = '\0';
954                 lcd_data(key);
955                 lastCharOperator = 1;
956             }
957         }
958         else if (key == '-' && expr_length == 0) { // Allow negative at start
959             expression[expr_length++] = key;
960             expression[expr_length] = '\0';
961             lcd_data(key);
962             lastCharOperator = 0; // Special case for negative at start
963         }
964     }
965 }
966 }
967 _delay_ms(100); // Debounce delay
968 }
969
970 return 0; // Never reached
971 }

```

The keypad matrix is given below

```
{'1', '2', '3', '+', '-', '*'},
{'4', '5', '6', '/', 'C', '='},
{'7', '8', '9', 'Q', 'M', 'D'},
{'0', 'e', 's', 'c', 't', 'S'}
```

Push Button Functions

- **Numeric Keys (0-9):** Used to enter numerical values.
- **Arithmetic Operators** (\div , \times , $-$, $+$): Perform basic arithmetic operations.
- **Exponential** : e^x : Calculates the value of e raised to the power x
- **Shift : S:** The shift button is used to toggle between trigonometric functions and inverse trigonometric functions of sin cos tan
- **Equal (=):** Evaluates the entered expression.
- **Differentiate (D):** Computes the derivative of an entered function.
- **Quadratic Solver : Q:** Quadratic equation solver it takes values a2 a1 a0 from the user and computes the roots of the quadratic equation
 - If $a_2 = 0$, it shows not a quadratic
 - If $\Delta < 0$, it shows no real roots
 - It shows Root 1 in line 1 and Root 2 in line 2
- **Differentiator : D:** It computes the derivative of a function f(x) taking input f(x) from the user
 - It also identifies operators + - * /
 - the number after x is consider as its power until any operator is recognised
- **Matrix Operator : M:** It takes two 2x2 matrices A and B and gives outputs addition and multiplication
 - It takes A and B matrices as inputs
 - Then it asks the user 3:A+B 4:AXB, after choosing an option
 - It displays the output matrix on the lcd

5 Results

1. Open terminal and go to your working directory

```
cd /sdcard/cprog/src
```

2. Write you code calculator.c inside src

```
nvim calculator.c
```

3. Execute the avr-gcc code using the below command, which creates .hex file

```
avr-gcc -mmcu=atmega328p -Os -o calculator.elf calculator.c &&
avr-objcopy -O ihex calculator.elf calculator.hex
```

4. Copy that .hex file into ArduinoDroid directory

```
cp calculator.hex /sdcard/ArduinoDroid/precompiled
```

5. Upload the precompiled code to the arduino using USB

