Implementation of Scientific Calculator Functions on Microcontroller using C

Krishna Patil, Nara Prajwal
Department of Electrical Engineering
IIT Hyderabad
Email: ee24btech11036@iith.ac.in, ee24btech11051@iith.ac.in

***Abstract*—This paper presents the implementation of scientific calculator functions such as trigonometric, logarithmic, exponential, power, factorial, and expression parsing using the C programming language on a microcontroller platform. Various numerical algorithms such as Runge–Kutta (RK4), series expansions, iterative methods, and parsing algorithms have been employed. Each function is discussed with its algorithmic basis followed by its C implementation.**

***Index Terms*—Scientific Calculator, Microcontroller, Runge–Kutta Method, Quake III Algorithm, Shunting Yard Algorithm, C Programming**

I. Introduction

The design and implementation of a scientific calculator on microcontrollers requires efficient mathematical algorithms due to limited hardware resources. This paper demonstrates the use of numerical methods such as RK4 for solving differential equations to compute trigonometric and exponential functions, iterative methods for logarithms, the Quake III inverse square root algorithm for efficient computations, and the Shunting Yard algorithm for expression parsing.

II. Software Implementation

Instead of using "general expansion methods" (like Taylor/Maclaurin series with many terms), we reformulated each function as a differential equation. The algorithms used to calculate the values are RK4 (for ODEs), Newton–Raphson (for refinements like inverse sqrt).

A. Supported Functions

TABLE I
Supported Functions and Operations

Category	Functions / Operations
Trigonometric	$\sin(x), \cos(x), \tan(x)$
Inverse Trigonometric	$\arcsin(x), \arccos(x), \arctan(x)$
Exponential / Logarithmic	$e^x, \ln(x)$
Power / Root	x^y, $\frac{1}{\sqrt{x}}$
Factorial	$n!$
Basic Arithmetic	$+, -, \times, \div$
Constants	π, e
Input Symbols	Digits 0–9, decimal point, parentheses $(,)$

B. Runge–Kutta Method (RK4)

The fourth-order Runge–Kutta method ([1], [2]) is a widely used numerical technique for solving ordinary differential equations of the form:

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0 \tag{1}$$

It estimates the solution by computing weighted averages of slopes at multiple points within the interval. Given a step size h, the next value is computed as:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2}$$

where

$$k_1 = hf(x_n, y_n), \tag{3}$$
$$k_2 = hf(x_n + h/2, y_n + k_1/2), \tag{4}$$
$$k_3 = hf(x_n + h/2, y_n + k_2/2), \tag{5}$$
$$k_4 = hf(x_n + h, y_n + k_3). \tag{6}$$

This method achieves $O(h^4)$ accuracy while maintaining computational efficiency. In this work, RK4 is applied to trigonometric and exponential functions by reformulating them as differential equations (e.g., $\sin(x)$ from $y'' + y = 0$, $\exp(x)$ from $y' = y$).

C. Trigonometric Functions

1) Sine Function: The sine function [3] is computed by solving the second-order differential equation:

$$\frac{d^2y}{dx^2} + y = 0, \quad y(0) = 0, \quad y'(0) = 1 \tag{7}$$

using the fourth-order Runge–Kutta method.

System:

$$\frac{dy}{dx} = z, \quad \frac{dz}{dx} = -y, \quad y(0) = 0,\ z(0) = 1 \tag{8}$$

RK4 step values:

$$k_1 = hz_n, \tag{9}$$
$$l_1 = -hy_n, \tag{10}$$
$$k_2 = h\left(z_n + \tfrac{l_1}{2}\right), \tag{11}$$
$$l_2 = -h\left(y_n + \tfrac{k_1}{2}\right), \tag{12}$$
$$k_3 = h\left(z_n + \tfrac{l_2}{2}\right), \tag{13}$$
$$l_3 = -h\left(y_n + \tfrac{k_2}{2}\right), \tag{14}$$
$$k_4 = h(z_n + l_3), \tag{15}$$
$$l_4 = -h(y_n + k_3), \tag{16}$$
$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{17}$$
$$z_{n+1} = z_n + \tfrac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4). \tag{18}$$

2) Cosine Function: The cosine function [3] is computed by solving the second-order differential equation:

$$\frac{d^2y}{dx^2} + y = 0, \quad y(0) = 1, \quad y'(0) = 0 \tag{19}$$

The RK4 recursion is identical to the sine case, but with different initial conditions.

3) Tangent Function: Tangent [3] is implemented as the ratio of sine and cosine.

D. Inverse Square Root

Quake III Algorithm: The method obtains an initial approximation by manipulating the IEEE 754 floating-point representation of x, then refines it using Newton-Raphson iterations.

1) Bit manipulation with a "magic constant" ($0x5f3759df$) produces an initial guess y_0.
2) A single Newton–Raphson iteration dramatically improves accuracy.
3) An optional second iteration yields nearly full precision.

Newton–Raphson Refinement: We want to approximate

$$y = x^{-\frac{1}{2}}. \tag{20}$$

Define

$$f(y) = \frac{1}{y^2} - x, \qquad f'(y) = -\frac{2}{y^3}. \tag{21}$$

Applying Newton–Raphson,

$$y_{k+1} = y_k - \frac{f(y_k)}{f'(y_k)} \tag{22}$$
$$= y_k - \frac{\frac{1}{y_k^2} - x}{-\frac{2}{y_k^3}} \tag{23}$$
$$= y_k + \frac{y_k}{2}\left(1 - xy_k^2\right) \tag{24}$$
$$= y_k\left(\tfrac{3}{2} - \tfrac{1}{2}xy_k^2\right). \tag{25}$$

Thus one Newton iteration is simply

$$y \leftarrow y\left(1.5 - 0.5\,x\,y^2\right). \tag{26}$$

E. Inverse Trigonometric Functions

1) Arcsine: The ODE for the arcsine function [3] is ODE:

$$\frac{dy}{dx} = \frac{1}{\sqrt{1-x^2}}, \quad y(0) = 0 \tag{27}$$

RK4 step values:

$$k_1 = \frac{h}{\sqrt{1-x_n^2}}, \tag{28}$$
$$k_2 = \frac{h}{\sqrt{1-(x_n + h/2)^2}}, \tag{29}$$
$$k_3 = \frac{h}{\sqrt{1-(x_n + h/2)^2}}, \tag{30}$$
$$k_4 = \frac{h}{\sqrt{1-(x_n + h)^2}}, \tag{31}$$
$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{32}$$
$$\tag{33}$$

2) Arccosine: Use the identity [3]

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x) \tag{34}$$

3) Arctangent: The ODE for the arctan function [3] is ODE:

$$\frac{dy}{dx} = \frac{1}{1+x^2}, \quad y(0) = 0 \tag{35}$$

RK4 step values:

$$k_1 = \frac{h}{1+x_n^2}, \tag{36}$$
$$k_2 = \frac{h}{1+(x_n + h/2)^2}, \tag{37}$$
$$k_3 = \frac{h}{1+(x_n + h/2)^2}, \tag{38}$$
$$k_4 = \frac{h}{1+(x_n + h)^2}, \tag{39}$$
$$y_{n+1} = y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{40}$$

F. Logarithmic Functions

The natural logarithm can be obtained by solving the differential equation [3]

$$\frac{dy}{dx} = \frac{1}{x}, \quad y(1) = 0 \tag{41}$$

which has the exact solution $y(x) = \ln(x)$. Using the RK4 method, we can approximate $\ln(x)$ by integrating this ODE from $x = 1$ to the target value.

RK4 step values:

$$k_1 = \frac{h}{x_n}, \tag{42}$$

$$k_2 = \frac{h}{x_n + h/2}, \tag{43}$$

$$k_3 = \frac{h}{x_n + h/2}, \tag{44}$$

$$k_4 = \frac{h}{x_n + h}, \tag{45}$$

$$y_{n+1} = y_n + \frac{h}{6}\left(\frac{1}{x_n} + \frac{4}{x_n + h/2} + \frac{1}{x_n + h}\right). \tag{46}$$

G. Exponential Functions

1) Power Function: The power function $y = x^w$ [3] satisfies

$$\frac{dy}{dx} = \frac{w}{x}y, \quad y(1) = 1 \tag{47}$$

RK4 step values:

$$k_1 = h \cdot \frac{w}{x_n} y_n, \tag{48}$$

$$k_2 = h \cdot \frac{w}{x_n + \frac{h}{2}} \left(y_n + \tfrac{k_1}{2}\right), \tag{49}$$

$$k_3 = h \cdot \frac{w}{x_n + \frac{h}{2}} \left(y_n + \tfrac{k_2}{2}\right), \tag{50}$$

$$k_4 = h \cdot \frac{w}{x_n + h} \left(y_n + k_3\right), \tag{51}$$

$$y_{n+1} = y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{52}$$

H. e^x Function

```
unsigned long long factorial(int n) {
  if (n==0) return 1;
  return n*factorial(n-1);
}
```

Listing 1. C Implementation of Factorial Function

I. Expression Parsing

The Shunting Yard algorithm is employed for converting infix expressions into postfix form for evaluation.

J. Overview

The **Shunting Yard Algorithm**, proposed by Edsger Dijkstra, is a method for converting mathematical expressions in infix notation into postfix notation (Reverse Polish Notation, RPN). It relies on two structures:

- A **stack** for operators/functions
- An **output queue** for the final postfix expression

K. Algorithm Steps

1) Initialize an empty stack and output queue.
2) For each token:
   a) Numbers $\rightarrow$ add to output.
   b) Functions $\rightarrow$ push to stack.
   c) Operators $\rightarrow$ pop higher/equal precedence operators from stack to output (respecting associativity), then push current operator.
   d) "(" $\rightarrow$ push to stack.
   e) ")" $\rightarrow$ pop to output until matching "(" is found, discard brackets.
3) After processing, pop remaining operators to output.

L. Operator Precedence

TABLE II
OPERATOR PRECEDENCE AND ASSOCIATIVITY

Operator	Precedence	Associativity
$\vee$ (power)	4	Right
$*$, $/$	3	Left
$+$, $-$	2	Left

M. Example

For the expression $3 + 4 * 2/(1 - 5)$, the Shunting Yard Algorithm yields the postfix form:

$$3\ 4\ 2\ *\ 1\ 5\ -\ /\ + \tag{53}$$

Algorithmic Steps :

TABLE III
SHUNTING YARD EXAMPLE FOR $3 + 4 * 2/(1 - 5)$

Token	Action	Stack	Output
3	Add to output		3
+	Push to stack	+	3
4	Add to output	+	3 4
*	Push to stack	+ *	3 4
2	Add to output	+ *	3 4 2
/	Pop *, push /	+ /	3 4 2 *
(	Push to stack	+ / (	3 4 2 *
1	Add to output	+ / (	3 4 2 * 1
-	Push to stack	+ / (-	3 4 2 * 1
5	Add to output	+ / (-	3 4 2 * 1 5
)	Pop until (	+ /	3 4 2 * 1 5 -
End	Pop stack		3 4 2 * 1 5 - / +

N. Reverse Polish Notation (RPN)

Reverse Polish Notation, also known as postfix notation, is a way of writing mathematical expressions in which operators follow their operands. Unlike infix notation (e.g., $3 + 4$), RPN eliminates the need for parentheses by unambiguously encoding operator precedence.

- Example (infix): $3 + 4$
- Equivalent RPN: $3\ 4\ +$

This simplicity makes RPN especially suitable for evaluation using a stack-based algorithm.

O. Evaluating RPN

Evaluation of RPN uses a stack:

1) Scan tokens left to right.
2) Push numbers on the stack.
3) For operators, pop required operands, apply operation, push result.

For the above example, evaluating the postfix expression gives the result 1 as expected .

P. Complexity and Applications

The algorithm runs in $O(n)$ time, where n is the number of tokens, since each token is processed once. Combined with RPN evaluation, it provides an efficient method for handling expressions in calculators, compilers, and symbolic computation systems.

Q. Function Handling

The Shunting Yard Algorithm can also be used to implement functions such as $\sin(x)$, $\cos(x)$, and others. Function handling follows these rules:

1) When a function token is encountered, it is pushed onto the operator stack.
2) Arguments (possibly separated by commas) are processed as sub-expressions in the usual way.
3) When the closing parenthesis ")" is reached:
   - Operators are popped from the stack to the output queue until the matching "(" is found.
   - The function token itself is then moved from the stack to the output queue.

As a result, function calls are correctly represented in postfix form. For example:

$$\sin(x) \quad \rightarrow \quad x \ \sin, \tag{54}$$

R. Conclusion

The Shunting Yard Algorithm, together with RPN evaluation, ensures efficient parsing and evaluation of mathematical expressions. It respects operator precedence, associativity, and bracket handling, making it a cornerstone in expression processing across computing applications.

S. Code Repository

The complete C implementations of the algorithms discussed in this paper are available at: https://github.com/gadepall/calculator/tree/main/codes

```
codes/
|-- ShuntingYard.c
|-- inv_sq_root.c
|-- inv_trig_func.c
|-- log.c
|-- power_func.c
|-- trig_func.c
```

Listing 2. Repository Structure in codes/

III. Hardware Implementation

References

[1] B. S. Grewal, *Higher Engineering Mathematics*, 43rd ed. New Delhi, India: Khanna Publishers, 2014.

[2] E. Kreyszig, *Advanced Engineering Mathematics*, 10th ed. Hoboken, NJ, USA: John Wiley & Sons, 2011.

[3] National Council of Educational Research and Training (NCERT), *Mathematics: Textbook for Class XII*, Part 1 and 2, New Delhi, India: NCERT, 2006.