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. Methodology

A. Runge–Kutta Method (RK4)

The fourth-order Runge–Kutta method 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 $y' = y$).

B. Shunting Yard Algorithm

The Shunting Yard algorithm, introduced by Edsger Dijkstra, provides an efficient method for parsing mathematical expressions written in infix notation into postfix (Reverse Polish Notation, RPN). It uses a stack-based approach to handle operator precedence and associativity.

- **Operands (numbers, variables):** sent directly to the output queue.
- **Operators:** pushed onto the operator stack, considering precedence and associativity.
- **Parentheses:** used to control order of operations. A right parenthesis triggers popping from the stack until the matching left parenthesis is encountered.

The postfix expression is then evaluated using a simple stack machine: operands are pushed, and operators pop the required number of operands, perform the operation, and push the result back. This makes it highly efficient for implementation in low-resource systems such as microcontrollers.

C. Quake III Fast Inverse Square Root

Algorithm: Bit-level hack and Newton–Raphson iteration for approximating $\frac{1}{\sqrt{x}}$.

This algorithm, popularized by the Quake III Arena game engine, provides an extremely fast approximation of the inverse square root using integer bit manipulations followed by one Newton–Raphson iteration. Despite its approximate nature, it delivers high accuracy with a fraction of the computational cost of traditional methods. In the context of this work, it is particularly useful for functions such as the arcsine, where the integrand requires repeated evaluation of $\frac{1}{\sqrt{1-x^2}}$.

III. Trigonometric Functions

A. Sine Function

The sine function 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.

```c
#define PI 3.14159265358979323846
#define H 0.05

double sine(double x_target) {
  double x=0, y=0, z=1;
  double k1,k2,k3,k4,l1,l2,l3,l4;
  int k=(int)(x_target/PI);
  double angle=fmod(x_target,2*PI);

  while (x<angle) {
    if (x+H>angle) H=angle-x;
    k1=H*z; l1=-H*y;
    k2=H*(z+l1/2.0); l2=-H*(y+k1/2.0);
    k3=H*(z+l2/2.0); l3=-H*(y+k2/2.0);
    k4=H*(z+l3); l4=-H*(y+k3);
    y+=(k1+2*k2+2*k3+k4)/6.0;
    z+=(l1+2*l2+2*l3+l4)/6.0;
    x+=H;
  }
  return y;
}
```

Listing 1. C Implementation of Sine Function

B. Cosine Function

Cosine can be derived similarly with initial conditions $y(0) = 1$, $y'(0) = 0$.

```c
double cosine(double x_target) {
  double x=0, y=1, z=0;
  double k1,k2,k3,k4,l1,l2,l3,l4;
  int k=(int)(x_target/PI);
  double angle=fmod(x_target,2*PI);

  while (x<angle) {
    if (x+H>angle) H=angle-x;
    k1=H*z; l1=-H*y;
    k2=H*(z+l1/2.0); l2=-H*(y+k1/2.0);
    k3=H*(z+l2/2.0); l3=-H*(y+k2/2.0);
    k4=H*(z+l3); l4=-H*(y+k3);
    y+=(k1+2*k2+2*k3+k4)/6.0;
    z+=(l1+2*l2+2*l3+l4)/6.0;
    x+=H;
  }
  return y;
}
```

Listing 2. C Implementation of Cosine Function

C. Tangent Function

Tangent is implemented as the ratio of sine and cosine:

```c
double tangent(double x) {
  return sine(x)/cosine(x);
}
```

Listing 3. C Implementation of Tangent Function

D. Inverse Trigonometric Functions

1) Arcsine: **Algorithm:** Integrate

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

using the RK4 method.

```c
double arcsin(double x) {
    if (x < -1 || x > 1) return 0;

    double x0 = 0.0, y = 0.0;
    int steps = x >= 0 ? (int) (x / H): (int) (-x / H);
    double step_dir = (x >= 0) ? 1 : -1;

    for (int i = 0; i < steps; i++) {
        double k1 = H * fast_inv_sqrt(1 - x0*x0);
        double k2 = H * fast_inv_sqrt(1 - (x0 + step_dir*H/2)*(x0 + step_dir*H/2));
        double k3 = H * fast_inv_sqrt(1 - (x0 + step_dir*H/2)*(x0 + step_dir*H/2));
        double k4 = H * fast_inv_sqrt(1 - (x0 + step_dir*H)*(x0 + step_dir*H));

        y += step_dir * (k1 + 2*k2 + 2*k3 + k4) / 6;
        x0 += step_dir * H;
    }

    return y;
}
```

Listing 4. C Implementation of Arcsine Function

2) Arccosine: **Algorithm:** Use the identity

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

```c
double arccos(double x){
    return ((PI/2) - arcsin(x));
}
```

Listing 5. C Implementation of Arccosine Function

3) Arctangent: **Algorithm:** Integrate

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

using the RK4 method.

```c
double arctan(double x) {
    double x0 = 0.0, y = 0.0;
    int steps = x >= 0 ? (int) (x / H): (int) (-x / H);
    double step_dir = (x >= 0) ? 1 : -1;

    for (int i = 0; i < steps; i++) {
        double k1 = H / (1 + x0*x0);
        double k2 = H / (1 + (x0 + H/2)*(x0 + H/2));
        double k3 = H / (1 + (x0 + H/2)*(x0 + H/2));
        double k4 = H / (1 + (x0 + H)*(x0 + H));

        y += step_dir * (k1 + 2*k2 + 2*k3 + k4) / 6;
        x0 += step_dir * H;
    }

    return y;
}
```

Listing 6. C Implementation of Arctangent Function

IV. Logarithmic Functions

The natural logarithm can be obtained by solving the differential equation

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

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.

```c
double logarithm(double x_target) {
  double x = 1.0, y = 0.0;
  double h = 0.001; // step size
  double k1, k2, k3, k4;

  while (x < x_target) {
    if (x + h > x_target) h = x_target - x;
    k1 = h * (1.0 / x);
    k2 = h * (1.0 / (x + h/2.0));
    k3 = h * (1.0 / (x + h/2.0));
    k4 = h * (1.0 / (x + h));
    y += (k1 + 2*k2 + 2*k3 + k4) / 6.0;
    x += h;
  }
  return y;
}
```

Listing 7. C Implementation of Logarithm Function using RK4

V. Exponential Functions

A. Power Function

Algorithm: Solve

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

with RK4.

```c
double pow(double x, double w) {
  if (x == 0) return 0;
  if (w == 0) return 1;

  double x0 = 1.0;
  double y = 1.0;  // y(1) = 1

  int steps = (int)((x - x0) / H);
  if (x < 1.0) {
    steps = (int)((x0 - x) / H);
    steps = -steps;  // Negative steps for x < 1
  }

  for (int i = 0; i < steps; i++) {
    double k1 = H * (w * y / x0);
    double k2 = H * (w * (y + 0.5 * k1) / (x0 + 0.5 * H));
    double k3 = H * (w * (y + 0.5 * k2) / (x0 + 0.5 * H));
    double k4 = H * (w * (y + k3) / (x0 + H));

    y += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    x0 += H;
  }

  double remaining = x - x0;
  if (remaining > 0) {
    double k1 = remaining * (w * y / x0);
    double k2 = remaining * (w * (y + 0.5 * k1) / (x0 + 0.5 * remaining));
    double k3 = remaining * (w * (y + 0.5 * k2) / (x0 + 0.5 * remaining));
    double k4 = remaining * (w * (y + k3) / (x0 + remaining));

    y += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
  }

  return y;
}
```

Listing 8. Power Function Implementation

B. e^x Function

```c
#define EULER_CONST 2.718281828459045

double exponential(double x_target) {

y = pow(EULER_CONST, x_target)
  }
  return y;
}
```

Listing 9. C Implementation of Exponential Function

C. Factorial Function

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

Listing 10. C Implementation of Factorial Function

VI. Expression Parsing

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

```c
int mode = 0;

typedef struct {
  char data[MAX_SIZE];
  int top;
} Stack;

void push(Stack *s, char val) {
  if (s->top < MAX_SIZE - 1) {
    s->data[++(s->top)] = val;
  }
}

char pop(Stack *s) {
  return (s->top >= 0) ?
    s->data[(s->top)--] : '\0';
}

char peek(Stack *s) {
  return (s->top >= 0) ? s->data[s->top] :
    '\0';
}
```

```c

int precedence(char op) {
  switch (op) {
    case '+': case '-': return 1;
    case '*': case '/': return 2;
    case '^': return 3;
    default: return 0;
  }
}

int isFunction(char c) {
  return (c == 's' || c == 'c' || c == 't' || c == 'e' || c == 'l' || c == 'z' || c == 'y' || c == 'x' || c == 'q');
}

// Convert infix to postfix
void shunting_yard(const char *expr, char *output) {
  Stack operators = {.top = -1};
  int j = 0;
  for (int i = 0; expr[i] != '\0'; i++) {
    if (isdigit(expr[i]) || expr[i] == '.') {
      while (isdigit(expr[i]) || expr[i] == '.') {
        output[j++] = expr[i++];
      }
      output[j++] = ' ';
      i--;
    } else if (isFunction(expr[i])) {
      push(&operators, expr[i]);
    } else if (expr[i] == '(') {
      push(&operators, expr[i]);
      unmatched_brackets++;
    } else if (expr[i] == ')') {
      while (peek(&operators) != '(') {
        output[j++] = pop(&operators);
        output[j++] = ' ';
      }
      unmatched_brackets--;
      pop(&operators); // Remove '('
      if (isFunction(peek(&operators))) {
        output[j++] = pop(&operators);
        output[j++] = ' ';
      }
    } else if (strchr("+-*/^!", expr[i])) {
      while (operators.top != -1 && precedence(peek(&operators)) >= precedence(expr[i])) {
        output[j++] = pop(&operators);
        output[j++] = ' ';
      }
      push(&operators, expr[i]);
    }
  }
  while (operators.top != -1) {
    output[j++] = pop(&operators);
    output[j++] = ' ';
  }
  output[j] = '\0';
}

// Evaluate postfix expression
double evaluate_rpn(const char *postfix) {
  double stack[MAX_SIZE];
  int top = -1;
  char token[20];
  int i = 0;
  while (*postfix) {
    if (isdigit(*postfix) || *postfix == '.') {
      sscanf(postfix, "%s", token);
      stack[++top] = atof(token);
      while (*postfix && *postfix != ' ') postfix++;
    } else if (strchr("!", *postfix)){
      double a = stack[--top];
//        stack[++top] = factorial(a);
    } else if (strchr("+-*/^", *postfix)) {
      double b = stack[top--];
      double a = stack[top--];
      switch (*postfix) {
        case '+': stack[++top] = a + b; break;
        case '-': stack[++top] = a - b; break;
        case '*': stack[++top] = a * b; break;
        case '/': stack[++top] = a / b; break;
        case '^': stack[++top] = pow(a, b); break;
      }
    } else if (isFunction(*postfix)) {
      double a = stack[top--];
      switch (*postfix) {
        case 's': stack[++top] = sin(a); break;
        case 'c': stack[++top] = cos(a); break;
        case 't': stack[++top] = tan(a); break;
        case 'e': stack[++top] = exp(a); break;
        case 'l': stack[++top] = ln(a); break;
        case 'z': stack[++top] = arcsin(a); break;
        case 'y': stack[++top] = arccos(a); break;
        case 'x': stack[++top] = arctan(a); break;
      }
    }
    postfix++;
  }
  return stack[top];
}
factorial : simple for loop
c
double factorial(double x){
  double result = 1;
  while (x > 1){
    result *= x;
    x -= 1;
  }
  return result;
```

Listing 11. Simplified Parser Implementation