1.1. Long Jumps

The **setjmp** standard library holds functions for preparing and performing long jumps; that is, jumps through function call chains.

setjmp.h

```
typedef void* jmp_buf[3];
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int value);
```

setjmp.c

```
#include <setjmp.h>
```

The **setjmp** function stores the return address, regular frame pointer, and variadic frame pointer in the **buf** static variable.

```
#ifdef __LINUX__
  int setjmp(jmp_buf buf) {
    void** rbp_pointer = register_rbp;
    buf[0] = rbp_pointer[0];
    buf[1] = rbp_pointer[1];
    buf[2] = rbp_pointer[2];
    return 0;
  }
```

The **longjmp** function resets the return address, regular frame pointer, and variadic frame pointer by looking up their values from the **buf** static variable by **setjmp** above. The last line jumps back to the return address of the previous **setjmp** call, with the return value given as a parameter to **longjmp**. In this way the previous **setjmp** call return zero and the following **longjmp** call returns the (presumably non-zero) value of its parameter.

```
  void longjmp(jmp_buf buf, int return_value) {
    register_ebx = return_value;
    register_rcx = buf[0];
    register_rdi = buf[2];
    register_rbp = buf[1];
    jump_register(register_rcx);
  }
#endif
```

The code for the Windows environment is similar to the Linux environment. The difference is that we use 16-bit registers rather than 32-bit and 64-bit registers.

```
#ifdef __WINDOWS__
  int setjmp(jmp_buf buf) {
    void** bp_pointer = register_bp;
    buf[0] = bp_pointer[0];
    buf[1] = bp_pointer[1];
    buf[2] = bp_pointer[2];
    return 0;
  }

  void longjmp(jmp_buf buf, int return_value) {
    register_bx = return_value;
    register_cx = buf[0];
    register_di = buf[2];
    register_bp = buf[1];
    jump_register(register_cx);
  }
#endif
```

The following code demonstrates how **setjmp** and **longjmp** can be used by jumping back through a function call chain.

setjmptest.c

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf buffer;

double inverse(double x);
double divide(double x, double y);

void main() {
  char* message;
  double x;

  printf("Please input a value: ");
  scanf("%lf", &x);

  if ((message = setjmp(buffer)) == 0) {
    printf("1.0 / %f = %f\n", x, inverse(x));
  }
  else {
    printf("%s\n", message);
  }
}

double inverse(double x) {
  return divide(1, x);
}
```

In case of a non-zero denominator, we return the quintet in a normal way and **setjmp** returns zero. In case of a zero denominator, we call **longjmp** with an error message. The address of the error message is then be returned by the call to **setjmp**, instead of zero.

```
double divide(double x, double y) {
  if (y != 0) {
    return x / y;
  }
  else {
    longjmp(buffer, "Division by Zero.");
    return 0;
  }
}
```

1.2. Mathematical Functions

The mathematical functions are implemented completely in C, they do not make system calls.

math.h

```
#ifndef __MATH_H__
#define __MATH_H__

#define PI 3.14159265358979323846264338
#define E  2.71828182845904523536028747

extern double exp(double x);
extern double log(double x);
extern double log10(double x);

extern double pow(double x, double y);
```

```
extern double ldexp(double x, int exponent);
extern double frexp(double x, int* exponent);

extern double sqrt(double x);
extern double modf(double x, double* integral);
extern double fmod(double x, double y);

extern double sin(double x);
extern double cos(double x);
extern double tan(double x);

extern double sinh(double x);
extern double cosh(double x);
extern double tanh(double x);

extern double asin(double x);
extern double acos(double x);
extern double atan(double x);
extern double atan2(double x, double y);

extern double floor(double x);
extern double ceil(double x);
extern double round(double x);
extern double fabs(double x);

#endif
```

1.2.1. Exponent and Logarithm Functions

The Sine, Cosine, Exponent, and Logarithm functions are calculated by iterative methods, and the rest of the functions call these functions.

math.c

```
#include <math.h>
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
```

The exponent function can be calculated be the following iterative formula for any values of x.

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \sum_{i=1}^{\infty} \frac{x^i}{i!}$$

Naturally, we cannot iterate indefinitely in the code. Instead, we break the iteration when the term is smaller than the EPSILON constant.

```
#define EPSILON   1e-9

double exp(double x) {
  double index = 1, term, sum = 1, faculty = 1, power = x;

  do {
    term = power / faculty;
    sum += term;
    power *= x;
    faculty *= ++index;
  } while (fabs(term) >= EPSILON);

  return sum;
}
```

The logarithm function can be calculated be the following formula for any values $0 < x < 2$, which gives $-1 < (x - 1) < 1$.

$$\ln x = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}(x-1)^i}{i}$$

To begin with, we must make sure that the input value is more than zero and less than two. Moreover, in order to keep the number of iterations down, the value shall not be too close to either zero or two. Therefore, we make sure the value holds the interval $\frac{1}{e} < x < 1$. We can do that by multiply or divide by e, and use the facts that

$$\ln xe^n = \ln x + \ln e^n = \ln x + n \ln e = \ln x + n \cdot 1 = \ln x + n$$

and

$$\ln x \sqrt[n]{e} = \ln x\, e^{-n} = \ln x + \ln e^{-n} = \ln x - \ln e^n = \ln x - n \ln e = \ln x - n \cdot 1 = \ln x - n$$

This gives that we can divide x with e until $x < 1$, and we can multiply x with e until $x > \frac{1}{e}$, as long as we keep track of the number of divisions and multiplications.

```
#define E_INVERSE (1 / E)

double log(double x) {
  if (x > 0) {
    int n = 0;
```

If x is more than zero and not exact one, we test whether x is more than one. If it is, we divide x with e until x is less than one.

```
    if (x > 1) {
      while (x > 1) {
        x /= E;
        ++n;
      }
    }
```

If x is less than $\frac{1}{e}$, we multiple e with e until it is more than more $\frac{1}{e}$. In both cases we keep track of the number of divisions and multiplications by updating n.

```
    else if (x < E_INVERSE) {
      while (x < E_INVERSE) {
        x *= E;
        --n;
      }
    }
```

When we have made sure that $\frac{1}{e} < x < 1$, we iterate with the formula $\ln x = \sum_{i=1}^{\infty} \frac{(-1)^{i+1}(x-1)^i}{i}$.

```
    { double index = 1, term, sum = 0, sign = 1,
             x_minus_1 = x - 1, power = x_minus_1;

      do {
        term = sign * power / index++;
        sum += term;
        power *= x_minus_1;
        sign *= -1.0;
      } while (fabs(term) >= EPSILON);
```

The result is the sum of the iteration plus the number of division and multiplication (n) before the iteration.

```
      return sum + n;
    }
  }
```

If x is less than zero, we report an error by setting **errno** domain error and return zero.

```
  else {
    errno = EDOM;
    return 0;
  }
}
```

The common logarithm, with base 10, is defined as follows:

$$\log_{10} x = \frac{\ln x}{\ln 10}$$

We define a constant value for $\ln 10$, which we divide $\ln x$ with.

```
#define LN_10 2.30258509299404568401799914

double log10(double x) {
  return log(x) / LN_10;
}
```

1.2.2. Power Functions

The power function x^y is defined for all real values $x > 0$ as follows:

$$x^y = \begin{cases} e^{y \ln x} & if\ x > 0 \\ 1 & if\ x = 0\ and\ y = 0 \\ 0 & if\ x = 0\ and\ y > 0 \end{cases}$$

If $x < 0$ and y is an integer value, then

$$x^y = \begin{cases} e^{y \ln(-x)} & if\ y \text{ is even} \\ -e^{y \ln(-x)} & if\ y \text{ is odd} \end{cases}$$

```
double pow(double x, double y) {
  if (x > 0)  {
    return exp(y * log(x));
  }
  else if ((x == 0) && (y == 0)) {
    return 1;
  }
  else if ((x == 0) && (y > 0)) {
    return 0;
  }
```

We test if y is an integer value by comparing its floor and ceiling values. If they are equal, the value is an integer.

```
  else if ((x < 0) && (floor(y) == ceil(y))) {
    long long_y = (long) y;
```

If the integer value of y modulo two is zero, it is even. Otherwise, it is odd.

```
    if ((long_y % 2) == 0) {
      return exp(y * log(-x));
    }
    else {
      return -exp(y * log(-x));
```

```
    }
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

The function **ldexp** is defined as $\mathrm{ldexp}\,(x, n) = x \cdot 2^n$.

```
double ldexp(double x, int n) {
  return x * pow(2, n);
}
```

The function **frexp** splits **x** into normalized fraction of **x**. The return value is within the interval from 0.5, inclusive, to 1, exclusive.

```
#define LN_2 0.69314718055994530941723212

static log2(double x) {
  return log(x) / LN_2;
}

double frexp(double x, int* p) {
  if (x != 0)  {
    int exponent = (int) log2(fabs(x));

    if (pow(2, exponent) < x) {
      ++exponent;
    }

    if (p != NULL) {
      *p = exponent;
    }

    return (x / pow(2, exponent));
  }
  else {
    if (p != NULL) {
      *p = 0;
    }

    return 0;
  }
}
```

1.2.1. Square Root

The square root r of x is for all $x > 0$ is iterative defined as:

$$\begin{cases} r_0 = 1 \\ r_{i+1} = \dfrac{r_i + \dfrac{x}{r_i}}{2} \end{cases}$$

```
double sqrt(double x) {
  if (x >= 0) {
    double root_i, root_i_plus_1 = 1;

    do {
      root_i = root_i_plus_1;
      root_i_plus_1 = (root_i + (x / root_i)) / 2;
    } while (fabs(root_i_plus_1 - root_i) >= EPSILON);
```

```
    return root_i_plus_1;
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

1.2.2. Modulo Functions

The **modf** function split **x** into a integral and fractional part. The integral is returned, and the fractional part is assigned the value that **p** points at, if it is not null. The integral and fractional values hold the same sign as the original value.

```
double modf(double x, double* p) {
  double abs_x = fabs(x),
  integral = (double) ((long) abs_x),
  fractional = abs_x - integral;

  if (p != NULL)  {
    *p = (x > 0) ? integral : -integral;
  }

  return (x > 0) ? fractional : -fractional;
}
```

The **fmod** function returns the floating-point remainder of **x** / **y**, with the same sign as **x**.

```
double fmod(double x, double y) {
  if (y != 0) {
    double remainder = fabs(x - (y * ((int) (x / y))));
    return (x > 0) ? remainder : -remainder;
  }
```

If **y** is zero, we have division by zero, we report a domain error and return zero.

```
  else {
    errno = EDOM;
    return 0;
  }
}
```

1.2.3. Trigonometric Functions

The sine function is defined for all x as follows:

$$\sin(x) = \sin(x + 2\pi n) \quad n \in \mathbb{Z}$$

$$\sin x = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!}$$

```
double sin(double x) {
  if (fabs(x) > (2 * PI)) {
    x = fmod(x, 2 * PI);
  }

  { double index = 1, term, sum = 0, sign = 1, power = x, faculty = 1;

    do {
      term = sign * power / faculty;
```

```
      sum += term;
      sign *= -1;
      power *= x * x;
      faculty *= ++index * ++index;
    } while (fabs(term) >= EPSILON);

    return sum;
  }
}
```

The cosine function is defined for all x as follows:

$$\cos(x) = \cos(x + 2\pi n) \quad n \in \mathbb{Z}$$

$$\cos(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i}}{(2i)!}$$

```
double cos(double x) {
  if (fabs(x) > (2 * PI)) {
    x = fmod(x, 2 * PI);
  }

  { double index = 0, term, sum = 0, sign = 1, power = 1, faculty = 1;

    do {
      term = sign * power / faculty;
      sum += term;
      sign *= -1;
      power *= x * x;
      faculty *= ++index * ++index;
    } while (fabs(term) >= EPSILON);

    return sum;
  }
}
```

The tangent function is defined as $\tan x = \frac{\sin x}{\cos x}$ for all x such as $\cos x \neq 0$.

```
double tan(double x) {
  double cos_of_x = cos(x);
  printf("cos(%f) = %f\n", x, cos(x));

  if (cos_of_x != 0) {
    return (sin(x) / cos_of_x);
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

1.2.4. Inverted Trigonometric Functions

The arcsine function is defined for all x such as $|x| \leq 1$ as follows:

$$\operatorname{asin}(1) = \frac{\pi}{2}$$

$$\operatorname{asin}(-x) = -\operatorname{asin}(x)$$

$$\operatorname{asin}(x) = \operatorname{atan}\left(\frac{x}{\sqrt{1 - x^2}}\right) \quad if \ \ |x| < 1$$

```
double asin(double x) {
  if (x == 1) {
```

```
    return PI / 2;
  }
  else if (x < 0) {
    return -asin(-x);
  }
  else if (x < 1) {
    return atan(x / sqrt(1 - (x * x)));
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

The arccosine function is defined for all x such as $|x| \leq 1$ as follows:

$$\text{acos}(0) = \frac{\pi}{2}$$

$$\text{acos}(-x) = \pi - acos(x)$$

$$\text{acos}(x) = \text{atan}\left(\frac{\sqrt{1-x^2}}{x}\right) \quad if\ 0 < |x| \leq 1$$

```
double acos(double x) {
  if (x == 0) {
    return PI / 2;
  }
  else if (x < 0) {
    return PI - acos(-x);
  }
  else if (x <= 1) {
    return atan(sqrt(1 - (x * x)) / x);
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

The arctangent function is defined for all $-1 \leq x \leq 1$ as follows:

$$\text{atan}(x) = -\text{atan}\,(-x) \quad for\ all\ x$$

$$\text{atan}(x) = \frac{\pi}{2} - \text{atan}\,\left(\frac{1}{x}\right) \quad if\ x > 0$$

$$\text{atan}(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{2i+1} \quad if\ 0 \leq x < 1$$

```
double atan(double x) {
  if (x < 0) {
    return -atan(-x);
  }
  else if (x > 1) {
    return PI / 2 - atan(1 / x);
  }
```

The number of iterations of the formula above is very high for values close to one. Therefore, we use the following formula for $\frac{1}{2} < x \leq 1$. The function argument of the right-hand call is always less than or equal to $\frac{1}{2}$, since the numerator is at most one and the denominator is at least two.

$$\text{atan}(x) = 2\,\text{atan}\left(\frac{x}{1+\sqrt{1+x^2}}\right)$$

```
  else if (x > 0.5) {
    return 2 * atan(x / (1 + sqrt(1 + (x * x))));
  }
```

We have finally reached the iterative formula. There is at most 30 iterations for values close to $1/2$, and less iteration for values closer to zero.

$$\text{atan}(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{2i+1} \quad if\ 0 \le x < 1$$

```
  else {
    double term, sum = 0, sign = 1, denominator = 1, product = x;

    do {
      term = sign * product / denominator;
      sum += term;
      sign = -sign;
      product *= x * x;
      denominator += 2;
    } while (fabs(term) >= EPSILON);

    return sum;
  }
}
```

The second arctangent function return arctangent of $\frac{x}{y}$, if $y \neq 0$.

```
double atan2(double x, double y) {
  if (y > 0) {
    return atan(x / y);
  }
  else if ((x >= 0) && (y < 0))  {
    return PI + atan(x / y);
  }
  else if ((x < 0) && (y < 0)) {
    return (-PI) + atan(x / y);
  }
  else if ((x > 0) && (y == 0))  {
    return PI / 2;
  }
  else if ((x < 0) && (y == 0))  {
    return (-PI) / 2;
  }
  else {
    errno = EDOM;
    return 0;
  }
}
```

1.2.5. Hyperbolic Trigonometric Functions

The hyperbolic sine function is defined for all x as follows:

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

```
double sinh(double x) {
  return (exp(x) - exp(-x)) / 2;
}
```

The hyperbolic cosine function is defined as follows:

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

```
double cosh(double x) {
  return (exp(x) + exp(-x)) / 2;
}
```

The hyperbolic tangent function is defined for all x as follows:

$$\tanh x = \frac{\sinh x}{\cosh x}$$

Not that we do not have to check for division by zero in this case, since $\cosh x$ cannot be zero.

```
double tanh(double x) {
  return sinh(x) / cosh(x);
}
```

1.2.6. Floor, Ceiling, Absolute, and Rounding Functions

The **floor** function returns the integer value closer to zero.

```
double floor(double x) {
  if (x < 0) {
    return -ceil(-x);
  }

  return (double) ((long) x);
}

double ceil(double x) {
  if (x < 0) {
    return -floor(-x);
  }

  return (double) ((long) (x + 0.999999999999));
}

double round(double x) {
  return (double) ((long) ((x < 0) ? (x - 0.5) : (x + 0.5)));
}
```

$$|x| = \begin{cases} -x & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$$

```
double fabs(double x) {
  return (x < 0) ? -x : x;
}
```