## 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;

}

}

## 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 3.1415926535897932384626433

#define E 2.7182818284590452353602874

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

### 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 .

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 , which gives .

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 . We can do that by multiply or divide by , and use the facts that

and

This gives that we can divide with until , and we can multiply with until , 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 is more than zero and not exact one, we test whether x is more than one. If it is, we divide with until is less than one.

if (x > 1) {

while (x > 1) {

x /= E;

++n;

}

}

If is less than , we multiple with until it is more than more . In both cases we keep track of the number of divisions and multiplications by updating .

else if (x < E\_INVERSE) {

while (x < E\_INVERSE) {

x \*= E;

--n;

}

}

When we have made sure that , we iterate with the formula .

{ 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 () before the iteration.

return sum + n;

}

}

If 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:

We define a constant value for , which we divide with.

#define LN\_10 2.3025850929940456840179914

double log10(double x) {

return log(x) / LN\_10;

}

### Power Functions

The power function is defined for all real values as follows:

If and is an integer value, then

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 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 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 .

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 0.6931471805599453094172321

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;

}

}

### Square Root

The square root of is for all is iterative defined as:

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;

}

}

### 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;

}

}

### Trigonometric Functions

The sine function is defined for all as follows:

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 as follows:

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 for all such as .

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;

}

}

### Inverted Trigonometric Functions

The arcsine function is defined for all such as as follows:

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 such as as follows:

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 as follows:

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 The function argument of the right-hand call is always less than or equal to , since the numerator is at most one and the denominator is at least two.

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 , and less iteration for values closer to zero.

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 , if .

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;

}

}

### Hyperbolic Trigonometric Functions

The hyperbolic sine function is defined for all as follows:

double sinh(double x) {

return (exp(x) - exp(-x)) / 2;

}

The hyperbolic cosine function is defined as follows:

double cosh(double x) {

return (exp(x) + exp(-x)) / 2;

}

The hyperbolic tangent function is defined for all as follows:

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

double tanh(double x) {

return sinh(x) / cosh(x);

}

### 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)));

}

double fabs(double x) {

return (x < 0) ? -x : x;

}