Programming in C with CCATSL

Historical Introduction

This manual consists of a short reference for the C language and a more substantial reference for version 2.1d of the CATAM software library, CCATSL: a collection of mathematical and graphical routines for use with a range of popular C compilers.

The CATAM projects, which originated in 1969 for Mathematics undergraduates in the University of Cambridge, with the object of encouraging computer exploration of aspects of the syllabus for the Mathematical Tripos. Originally, CATAM stood for Computer-Aided Teaching of *Applied* Mathematics, but this was extended several years ago to Computer-Aided Teaching of *All* Mathematics. It was pioneered by Dr Robert Harding of the Department of Applied Mathematics and Theoretical Physics (DAMTP); today it is supervised by the Computational Projects Assessors Committee, chaired by the Director Dr Nikolaos Nikiforakis of DAMTP.

Serious computer work begins in the second year: undergraduates become familiar with the C language, with the mathematical and graphical routines comprising CCATSL, and with networked computers. There are varied projects in the second year covering pure and applied, statistical and applicable mathematics. In the third year, the choice is very wide, and the projects relate directly to specific lecture courses in the Mathematical Tripos. These include fluid and solid mechanics, quantum theory, dynamics, general relativity, astrophysics, numerical methods, optimization, dynamical systems, number and group theory, algebra, analysis, statistics and probability. Most students now take the CATAM option, and it can contribute significantly towards the class of degree. The CATAM Software Library has been carefully designed to facilitate the programming of projects in C, removing much of the tedious work in equation solving, graphics, and screen layout. New users can begin with the high-level graphics routines where minimum knowledge is required, and gradually develop their expertise towards the lower levels which demand a more detailed understanding.

There is no argument today about the fundamental place of computer methods in mathematics. As is being demonstrated in research laboratories throughout the world, the combination of numerical methods with computer programming is bringing to life so much that was formerly intractable. Successful completion of CATAM projects implies the development of important programming and investigative skills of widespread value in industrial and commercial work, as well as in scientific research.

This version of the manual, in both printed form and online version (which can be accessed from the CATAM home web page, `http://www.maths.cam.ac.uk/catam`), was prepared for the Faculty of Mathematics by Giles Thompson, based on the two previous versions: the *Scientific Programmer's Toolkit* by M. H. Beilby, R. D. Harding and M. R. Manning, which is the primary reference for the chapter on Mathematical routines, and the *CATAM Manual* by John Evans.

Typographical Conventions

We will use a few typographical conventions throughout this manual. A `fixed-pitch font` will indicate text, possibly just single words, which would normally appear in a program listing, such as `printf`.
Function definitions follow a standard layout:

```
void CubicRootsCL (double a,
                   double b,
                   double c,
                   int *nroots,
                   double *r1,
                   double *r2,
                   double *r3);
```

Here we are presenting the definition of the function `CubicRootsCL`, which solves a cubic equation. The left-hand and middle columns give the return type (`void`) and the name of the function (`CubicRootsCL`) while the right-hand column gives the type of each argument. Note that some of the arguments to `CubicRootsCL` are declared with a `*` (see Section 1.4.5) indicating that `Cubic` will use `nroots`, `r1`, `r2` and `r3` to pass back information about the roots of the cubic; the values of `a`, `b` and `c` will not be changed by `CubicRootsCL`. After a definition like this comes a description of the meaning of all the arguments:

`a, b, c`	The coefficients in the cubic equation $x^3 + ax^2 + bx + c = 0$.
`nroots`	Pointer to a variable to hold the number of roots when `CubicRootsCL` returns.
`r1, r2, r3`	Pointers to variables to hold the roots, in non-decreasing order.

From this information we can see that a typical call of `CubicRootsCL` might look like

```
CubicRootsCL(0,-1, 0, &n, &a, &b, &c);
```

which will solve $x^3 - x = 0$, by setting $n = 3$, $a = -1$, $b = 0$ and $c = 1$ to indicate the three roots.

Contents

Chapter 1

Programming in C

1.1 Introduction

This chapter is a partial reference manual for the C language and the standard C library, concentrating on the parts likely to be useful to mathematics undergraduates writing programs for the CATAM projects, using computers on the University PWF, and emphasizing the differences between C and Pascal. The code extracts in the chapter do not use the CCATSL functions but can be compiled with the `gcc` software supplied with CCATSL—see the CCATSL on-line info.

Although some features of the PWF setup will not be shared with C compilers in general, this will be hilighted in the text, and most of the information will probably be useful to students programming in C on other systems.

This is not a tutorial, and users new to C and to CCATSL are strongly advised to seek out the 'Green Book', *Learning to use C and the CATAM Software Library*, by Dr C. D. Warner, which gives a gentle introduction to both the C language and the CCATSL library.

Less importance aspects of the language will not be covered, and we will refer the reader to a proper reference like:

> *The C Programming Language* by B. W. Kernighan and D. M. Ritchie, Prentice Hall.

Another book on C which may be helpful is:

> *A Book on C* by Al Kelley and Ira Pohl, Addison-Wesley, fourth edition.

1.2 Some examples

The basic language elements are most easily illustrated with examples. Our first example just prints a friendly greeting:

```
/*   /examples/chapter1/hello.c  */
#include <stdio.h>

int main(void) {
  printf("Hello from CATSL!\n");
  return 0;
}
```

The first line is a comment; its starts with the `/*` and continues until the next `*/` is found (it could extend over several lines). The next line gives access to the standard input-output library, which is necessary to use the `printf` function. The next line is blank for clarity (blank lines and spaces can be inserted liberally to make programs easier to read).

The rest of the file defines a function called `main`. This is the function which will be called when the program starts executing. Observe that `main` is defined as returning an integer value (the `int` keyword), it takes no arguments (the `(void)`) and that the body of the function is contained between curly brackets, `{` and `}`. Inside `main` we have two statements: first we print the greeting, which ends with a newline character (the `\n` is interpreted as a newline), and then we return a value of 0 (it is normal for `main` to return 0 if the program doesn't encounter any errors).

A few important rules are obeyed by our program:

- each statement must be terminated with a semi-colon,
- strings are enclosed in double-quotes,
- C is case-sensitive (replace `printf` with `Printf` and the program will not work),
- even though `printf` is a function (it returns an `int`), we can ignore its return value. C lets you ignore the return from any function; used in this way, `printf` behaves much like a procedure in Pascal.

Time for a more complex example. Our next program prompts the user for a number, and then computes the volume of the sphere with that radius.

```
/*  Compute the volume of a sphere, given the radius
    /examples/chapter1/sphere.c */
#include <stdio.h>
#define PI 3.14159265358979323846

int main(void) {
  float r, vol;
  do {
    printf("Enter radius (0 to quit): ");
    scanf("%f",&r);
    vol = 4.0*PI*r*r*r/3.0;
    printf("radius = %f\n volume = %f\n",r,vol);
  } while (r > 0);
  return 0;
}
```

Since standard C does not define π, we do so ourselves with a `#define` statement. In `main` we declare the variables `r` and `vol` as storing `float`s (a kind of floating-point number). Variables must be declared before they are used, and at the start of a 'compound-statement' (the curly brackets define the start and end of a compound-statement).

We use a `do-while` loop to let the user calculate the volume of several spheres without having to re-run the program. Within the loop we prompt the user for the radius with `printf`, read it in with `scanf` (we'll describe `scanf` in more detail shortly), calculate the volume and write out the answer. Note that assignment is a simple `=` in C, rather than `:=` as in Pascal.

To read in the radius, we use the function `scanf`. The first argument, the string, tells `scanf` how to interpret what the user types (the `%f` indicates we want a single floating-point number and asks for the number to be stored in a `float` variable), while the second, `&r` tells `scanf` to put the answer in `r`.

To write out the answer, we again use `printf`, but this time with three arguments. The first is the 'format string', which is mostly just copied to the screen (note the `\n` newline characters), except for the `%f` sequences which `printf` replaces, in turn, with the value of the other arguments: the first `%f` is replaced with the value of `r`, and the second with the value of `vol`.

The entire compound-statement between the curly brackets after the `do` keyword is repeated until the `while` condition fails, i.e. until the user enters a non-positive radius (note that the program will still write out the radius and volume when this happens because the `while` condition is only checked at the bottom of the compound-statement). Unlike Pascal, the brackets in the `while` statement around the continuation condition, `r > 0`, are required in C.

We finish with a more complex example, which contains almost everything required to a really useful C program.

```
/*  Use binary section to locate cube-roots
    /examples/chapter1/cube_root.c */
#include <stdio.h>

float f(float x) {
  return x*x*x;
}

int main(void) {
  float guess, f_guess;
  float cube;
  float lower_bound, upper_bound;
  int iter, max_iters;

  printf("Enter maximum number of iterations: ");
  scanf("%d", &max_iters);
  printf("Enter number to cube-root: ");
  scanf("%f", &cube);

  lower_bound = 0;
  upper_bound = 100;
  iter = 0;
  while (iter <= max_iters) {
    guess = 0.5*(lower_bound + upper_bound);
    f_guess = f(guess);
    printf("Iteration %d: ",iter);
    printf("guess = %f, f(guess) = %f\n",guess,f_guess);
    if (f_guess < cube) {
      lower_bound = guess;
    } else {
      upper_bound = guess;
    }
    if (upper_bound - lower_bound < 5e-7) {
      /*  We may aswell stop here */
      return 0;
    }
    iter = iter+1;
  }
  return 0;
}
```

Most of this should be fairly self-explanatory. The most important difference is that we define a function `f` as well as `main`. This function takes a single `float` argument and returns a `float`, its cube. It is worth pointing out that as soon as the `return` statement in `f` executes, the function will return: any statements inside `f` after the return statement (here there happen to be none) would never be executed. In `main`, we declare the variables `iter` and `max_iters` to be `int`s (the normal integer type). These must be handled differently from `floats` in `scanf` and `printf` statements. Where with `float`s we used `%f`, `int`s need `%d`.

Rather than a `do-while` loop, here we have used a `while` loop. The only significant difference is that the continuation condition (`iter <= max_iters`) is tested before the compound-statement is executed, rather than at the end.

We also use an `if-else` construction to adjust the bounds according to whether the the cube root lies to the left or right side of the current guess.

1.3 Variables

1.3.1 Declarations and scope

The syntax of most variable declarations is simple: the type of the variable, followed a list of names, or names with array bounds:

```
int x;
int y[2], b, c[3];
double a[3][3][3];
char names[][80];
struct point x;
enum colour foreground;
int *buf, *k;  /* pointers have a special syntax,
                  buf and k are both int* (pointers-to-ints)  */
```

The name can contain any sequence of numbers, letters, or the underscore character _ but cannot start with a number. The ANSI C standard dictates that only the first six characters in a variable's name are significant, but this restriction is ignored by most C compilers. A variable can only be declared at the start of a compound-statement,

```
{ /*  curly brackets starting a compound-statement  */
  int   n_iters;
  float answer;

  /*  use n_iters and answer here  */

} /*  end of the compound-statement  */
```

and can be used anywhere inside the compound-statement (we say that the 'scope' of the variable is confined to the compound-statement), or outside of any compound-statement, when it can be accessed from any point below its declaration. In the latter case, the variables are frequently declared right at the top of the program, and are thus visible everywhere (so called 'global' variables):

```
int i;

int f(void) {
  /*  can use i here  */
}

int main(void) {
  /*  can use i here too  */
}
```

Variables can be initialized when they are declared:

```
int f(int k) {
  int n_0 = 23;  /*  fine  */
  int m_0 = k;
}
```

If you start a new compound-statement, for example with an `if-else` construction, you can use the opportunity to declare new variables, but the new variables will only be visible inside the new compound-statement, below the declaration.

```
int f(int k) {
  int n = 1;

  if (n == 1) { /*  we start a new compound-statement;
                    can now define more variables  */
    int m = 4;
    /*  can use n and m here  */
  } /*  end of the compound-statement  */
  /*  can't use m here, we're outside the
      compound-statement it's declared in */
}
```

If a variable is declared when there is already a variable in scope with the same name, the second declaration obscures the first:

```
{
  int n = 1;
  /* .. */
  {
    int p = n;  /*  ok, the only n in scope is declared above */
    int n;      /*  declare a new n, shadowing the one above  */
    n = 23;     /*  effects the n declared on the line above  */
  }
 /*  n is 1 here, and there is no p in scope  */
}
```

1.3.2 Automatic conversions and casts

Once variables have been declared, they can be manipulated and combined in various ways. Exactly which operators may be applied to a variable depends on its type: for example all the numerical types support the usual arithmetic operations (addition, multiplication etc.) but only integer types support the modulus operator, `%`.

Sometimes you need to convert from one type to another, for example when assigning an `int` to a floating-point type. It is clear what the result of such a conversion should be, and C is prepared to convert between most pairs of related types automatically:

```
int   a = 6;
float b = 3.5;
int   c;

c = b;  /*  ok, b gets rounded towards zero, so c=3  */
b = a;  /*  ok, b becomes the real number 6  */
```

To convert between types for which an automatic conversion does not exist, or to force a conversion at a convenient moment, use a cast:

```
int   n = 4;
int   m = 6;
float n_over_m;

n_over_m = n/m;            /*  yields 0  */
n_over_m = n/ (float)m;    /*  (cast m to a float) yields 0.66666 etc.  */
```

Without the cast, `n/m` will be evaluated using integer division (which returns the quotient, discarding the remainder). The cast forces `m` to be converted to a `float`, after which `n` is automatically converted to a floating-point type and the division is done floating-point.

1.3.3 Storage-class specifiers (advanced)

A variable declaration can also be preceded by a storage-class specifier: `auto`, `extern`, `register` or `static`. Here we will only describe `static`, and refer the reader to a more complete reference manual for descriptions of the others.

Variables declared at the start of a compound-statement are only visible inside that compound-statement. Normally, each time the thread of execution enters a compound-statement, space is allocated for these variables and, if necessary, they are initialized. When it leaves, the space occupied by the variables is reclaimed and the value held by the variable is lost. If the declaration is prefixed with `static`, the variable will not be destroyed when compound-statement is left, and its value will be preserved. For example:

```
float f(float x) {
  static int init = 0;
  if (init == 0) {
    /*  put setup code here  */
    init = 1;
  }
  /*  by now things will be setup  */
}
```

Here we have a function `f` which must do some complex setup when its called for the first time. By using a `static` variable, we can tell when this happens. `init` will be initialized to zero as the program is loaded, and the first time `f` is called the setup code will run, and set `init` to one. On subsequent calls to `f`, `init` will still be one and the setup code will not be repeated.

1.4 Types

1.4.1 Numerical types

C has a wide range of numerical types, of which the most useful are `int`, `float` and `double`. The ranges of the various types on the PWF system are given in the tables below. (Three types not shown are `signed short`, `signed int` and `signed long`, which are synonyms for `short`, `int`, and `long` respectively). To find out the ranges on other systems, use the code at the end of this section. On many systems, `float` quantities lack precision and `double` quantities are to be preferred. On the PWF all CCATSL functions require `double` floating-point quantities.

Literals

Integer literals, such as `2` and `-3425` can be entered in the normal way, and floating-point literals may be entered in a number of formats `.34`, `0.34`, `3.4e-1`, `3.4E-1` etc. (see also Section 1.4.2).

Operators

All the numerical types support the usual binary arithmetic operations, addition `+`, subtraction `-`, multiplication `*`, and division `/`, and also unary minus `-`. Note that dividing two integer types does not give a floating-point number, but another integer, the quotient. In particular, an expression like `n/2` when `n` is an integer type is treated as integer division. To get the correct (floating-point)

type name	size (bytes)	lower limit	upper limit
`unsigned char`	1	0	255
`signed char`	1	−128	127
`char`	1	−128	127
`unsigned short`	2	0	65535
`short`	2	−32768	32767
`unsigned int`	4	0	4294967295
`int`	4	−2147483648	2147483647
`unsigned long`	4	0	4294967295
`long`	4	−2147483648	2147483647

Table 1.1: The integer numerical types on the PWF system.

type name	size (bytes)	positive lower limit	positive upper limit	epsilon
`float`	4	1.175494e−38	3.402823e+38	1.192093e−07
`double`	8	2.225074e−308	1.797693e+308	2.220446e−16
`long double`	12	2.225074e−308	1.797693e+308	2.220446e−16

Table 1.2: The floating-point numerical types on the PWF system.

answer when `n` is odd, use `n/2.0` since the literal `2.0` will be interpreted as a floating-point number, and force floating-point division.

All numerical types also support the binary comparison operators, `<`, `>`, `<=`, `>=`, `==` (equality) and `!=` (non-equality).

Integer types support the modulus operator, `%` which returns the remainder when the left argument is divided by the right argument e.g. `4%3` evaluates to 1. Two warnings: the answer will always have the same sign as the left argument, and if the right hand side of the `%` symbol evaluates to zero, your program will crash.

Integer types also support bitwise operators: `&` (and), `|` (or), `^` (xor), `~` (not), `<<` (left shift) and `>>` (right shift). All of these operators are binary except `~` which is unary.

Overflow

When you assign one variable to another, it is possible that the value will overflow the range of the destination variable. For integer types it is common for the value to 'wrap round'. For floating-point types, you may obtain one of the special numbers `Infinity`, `-Infinity` or `NaN` (not-a-number). These are designed to behave as you'd expect in most calculations, for example `2+Infinity` evaluates to `Infinity`. They can be displayed using `printf`; you may also be able to detect them using `isnan`, `isinf` and `finite`, but these functions are not standard.

Determining the range of numerical types

The following code shows how to find the size, range and precision of the numerical types:

```
/*   /examples/chapter1/limits.c */
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main(void) {
```

```
  /*  a byte doesn't have to contain 8 bits (but usually does) */
  printf("1 byte contains %i bits\n", CHAR_BIT);

  /*  char may be signed or unsigned, and all three will always be 1 byte  */
  printf("unsigned char   %u byte,  from %12i to %12i\n",
         sizeof(unsigned char), 0, UCHAR_MAX);
  printf("signed char     %u byte,  from %12i to %12i\n",
         sizeof(signed char), SCHAR_MIN, SCHAR_MAX);
  printf("char            %u byte,  from %12i to %12i\n",
         sizeof(char), CHAR_MIN, CHAR_MAX);

  /*  short, int and long are signed  */
  printf("unsigned short  %u bytes, from %12hu to %12hu\n",
         sizeof(unsigned short), 0, USHRT_MAX);
  printf("signed short    %u bytes, from %12hi to %12hi\n",
         sizeof(signed short), SHRT_MIN, SHRT_MAX);
  printf("unsigned int    %u bytes, from %12u to %12u\n",
         sizeof(unsigned int), (unsigned int) 0, (unsigned int) UINT_MAX);
  printf("signed int      %u bytes, from %12i to %12i\n",
         sizeof(signed int), INT_MIN, INT_MAX);

  printf("unsigned long   %u bytes, from %12lu to %12lu\n",
         sizeof(unsigned long), (unsigned long) 0, ULONG_MAX);
  printf("signed long     %u bytes, from %12li to %12li\n",
         sizeof(signed long), LONG_MIN, LONG_MAX);

  /*   floating point types can store 0, may also be able to store
       strange numbers like +Infinity and  */
  printf("float           %u bytes, positive range from %12e to %12e\n",
         sizeof(float), FLT_MIN, FLT_MAX);
  printf("                         epsilon = %e\n", FLT_EPSILON);
  printf("double          %u bytes, positive range from %12e to %12e\n",
         sizeof(double), DBL_MIN, DBL_MAX);
  printf("                         epsilon = %e\n", DBL_EPSILON);

  /*   L in format string is not universally supported  */
  printf("long double     %u bytes, positive range from %12Le to %12Le\n",
         sizeof(long double), LDBL_MIN, LDBL_MAX);
  printf("                         epsilon = %Le\n", LDBL_EPSILON);
  return 0;
}
```

1.4.2 The `char` type

`char` is a numerical type (see Section 1.4.1), but its main use is to represent characters, not numerical work. C provides a wide range of character literals which return the numerical value of a character (usually the ASCII code). Character literals are enclosed in apostrophes, for example:

```
char c = 'H';
```

will assign to `c` the numerical value of the character `H`. Since `char` is an integer type and C will convert between numerical types quite readily, the following are perfectly legal

```
int i = 'H';
double f = 'c';

i = 'K' * 67.57;
```

but probably not very useful. C also defines character literals for less convenient characters, such as newline and the tab character (see Table 1.3) (these actually give an `int` rather than a `char`, but this difference is not normally important). The newline, horizontal and vertical tab and form-feed characters, together with the space character are known collectively as whitespace.

literal	meaning
`\a`	alert (bell)
`\b`	backspace
`\f`	page-break (form-feed)
`\n`	newline (line-feed)
`\r`	carriage-return
`\t`	horizontal tab
`\v`	vertical tab
`\"`	double-quote
`\'`	apostrophe
`\\`	backslash
`\`*ooo*	the character with numerical value *ooo* in octal
`\x`*hh...*	the character with numerical value *hh...* in hexadecimal

Table 1.3: Special character literals

1.4.3 Arrays and Strings

Arrays

C lets you declare arrays of any type (provided the size of the type is known), and to access the elements using the `[]` syntax:

```
int i[100];  /* an array of 100 integers */
char c[80];  /* an array of 80 characters */

i[23]  = 1;  /* set the 23rd element */
i[0]   = 1;  /* set the 0th element */
i[99]  = 1;  /* ok, set the last element */
i[100] = 1;  /* illegal - there is no 100th element */
```

Note that arrays in C are always indexed from zero, and that the number in square brackets in the declaration is the number of elements in the array, not the largest possible index. The example above will probably compile as if nothing was amiss, but the program may crash when it executes the last line.

Arrays can be initialized when they are declared; in this case you need not specify the size of the array:

```
int x[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
/*  now x[0]=1, x[1]=2, .. , x[9]=10  */
```

To pass arrays to, and use array in functions, see Section 1.8.4.

Strings

C has no special 'string' type. A C string is just an array of `char`, with the understanding that last character is zero (not the character literal `'0'`, but the character with numerical value zero). C provides a familiar notation for string literals: simply enclose the string in double-quotes, and C will add the zero-terminator by itself:

```
char c[] = "abcde";
/*  c contains 6 chars, 'a', 'b', 'c', 'd', 'e', 0  */
```

Normally you do not need to worry about the zero-terminator unless you have to know exactly how much space the string will take (see below for an example).

You may use any character literal (see Section 1.4.2) in a string literal:

```
char c[] = "Done.\n";
```

Operators on arrays and strings

Arrays themselves do not support assignment (though their elements may do so), and comparison operators `==`, `<`, etc. will not compare the elements (see Section 1.4.5 for an explanation of what actually happens):

```
int  x[4] = { 1, 2, 4 };
int  w[4] = { 1, 2, 4 };
char y[4] = { 1, 2, 4 };

x = y;       /*  illegal, attempt to assign to the array x  */
x[2] = y[2]; /*  fine, x[2] is an int and you can assign a char to an int  */
if (w == x) {
  /*  this condition is false: w and x are not equal,
      even though all their corresponding elements are  */
}
```

Other than `[]` to access elements, you can use the & (address-of) operator, `+` and `-`. These are described in Section 1.4.5.

Two-dimensional arrays

Two-dimensional and multi-dimensional arrays are treated as arrays-of-arrays, and unlike in Pascal, do not have any special indexing syntax:

```
int x[3][2] = { {1, 0}, {0, 1}, {6, 9} };
/*  access elements via x[0][0], x[0][1], x[1][0], ... , x[2][1] etc  */
```

As in Pascal, the elements of an array are organized in memory in the order shown above, with the last index incrementing first.

Arrays of strings are just arrays-of-arrays-of-`char`:

```
char c[][5] = { "cats", "dogs" };   /*  an array-of-array-of-5-chars  */

/*  note that we must leave room for the strings' terminating zero.
    (This example can be done more simply using pointers.)
    Now c[0] and c[1] are strings containing the names of pets  */

char d[][] =
  { "cats", "dogs" };   /*  illegal: char[] is a type of unknown size  */
```

1.4.4 Logical expressions

There is no boolean type in C (you cannot declare a variable to be boolean, or return a boolean from a function), but it does have a notion of a logical expression usually found in `if`, `for` and `while` statements.

```
if (a == 1) {
  /* .. */
}

while (row < n_rows) {
  /* .. */
}
```

Such expressions are normally formed using one of the comparison operators (`<`, `<=`, `==` (equality) `!=` (non-equality), `>=` and `>`). In fact, if `e` is an arbitrary expression which is to be interpreted as true of false, C will treat the expression as `e!=0` if `e` is a numeric type

```
if (a) {
  /*  do something  */
}
```

will be interpreted as

```
if (a != 0) {
  /*  so something  */
}
```

C will convert from a logical expression to a numerical type automatically using the convention: true becomes 1 (or 1.0 for floating-point types), false becomes 0 (or 0.0). Combined with the behavior described above, you can use any numerical type as if it were boolean, with the interpretation zero=false, non-zero=true.

```
int a;      /*  we'll use this as if it were boolean  */

a = (1<2);  /*  assigns 1 to a because the expression is true  */
```

(CCATSL defines a `boolean` type and `boolean` constants `true` and `false`.)

Logical operators

Logical expressions support operators analogous to 'and' and 'or' in Pascal. C provides: `&&` (and), `||` (or), `!` (not) together with `==` (equality or nxor) and `!=` (non-equality, or xor):

```
int a;

a = (1<2);  /*  assigns 1 to a  */
a = !a;     /*  a was one (true), hence a becomes zero (false)  */
/*  now a is zero  */
if (a > -1 && a < 1) {
   /*  the condition is true  */
}
```

1.4.5 Pointers

Pointers play a larger role in C than in Pascal, and although you may be able to avoid them by using arrays (Section 1.4.3), most programmers are likely to meet pointers eventually (if not actually use them).

Memory addresses

When a variable, such as an `int` or a `float` is stored in memory, it occupies a number of bytes. The *address* of a variable is a number (typically a large integer) identifying the byte where this storage starts. This number can be obtained for any variable using the address-of operator, `&`. If `v` is declared as a `char`, and you modify the byte with address `&v`, you will change the value of `v`.

Let's suppose that `&v` equals 100, and see how to change the value of `v` to `'A'`, using just the information that `&v` is 100. To do this, we must tell the compiler to interpret the bytes starting at address 100 as a `char`, and store `'A'` there. It is important to tell the compiler to store the `'A'` as a `char` since the `char 'A'` (which is likely to have the numerical value 65) will be stored in a totally different way from a `double` with value 65. This is where pointer types and the indirection operator `*` come in:

```
int main(void) {
  char v = 'B';  /*  set v to 'B' initially.  We'll change it below  */
  char *p;       /*  p has type pointer-to-char, will store the
                     address of a char  */
  p = (char*)100;

  *p = 'A';  /*  assign 'A' to the memory pointed to by p  */
  /*  if v were stored at address 100, v would now be the character 'A'  */
}
```

We declare `p` to have the type pointer-to-`char`, so that the compiler will treat `p` as holding the address of a `char`. We then use the indirection operator, `*` to modify 'the memory pointed to by `p`'. This code will only work if `v` really is stored at address 100 (which is unlikely). The following will always work:

```
char v = 'B';   /*  set v to 'B' initially.  We'll change it below.  */
char *p;

p = &v  /*  p now stores the address of v ('p points to v') */

*p = 'A';  /*  assign 'A' to the memory referred to by p  */
```

This example works because we can find out where the variable `v` is stored and then modify the memory directly. In most modern systems, a program is only allowed to modify memory specifically allocated to it, and even then it may be possible for a program to have some 'read-only' memory. You can assume that variables will not be stored in read-only memory unless you ask for it, and can get new blocks of (writable) memory at runtime using `malloc` (see `free` also). The compiler is quite at liberty to put literals in read-only memory however:

```
char *p = "This is a string literal";

*p = 'L';  /*  may crash  */
```

Here we assign to `p` the address of the first character in the string. (Why the compiler treats the assignment like this—even though the string is an array-of-`char` rather than a pointer—is explained below.) Then we try to modify the string by changing the first character to an 'L'. If the compiler put the literal in read-only memory, the program is going to crash.

Pointers and `scanf`

You are most likely to first meet pointers in conjunction with the function `scanf`, which reads characters from the keyboard. The idea is to pass to `scanf` the address of the variable where the input should be stored, and let `scanf` write the information straight into memory.

```
int my_integer_variable;
scanf("%d",&my_integer_variable);
```

`scanf` knows that we want an `int` (the `%d` asks for one) so it will take the second argument (which to `scanf` is just a large integer, but is actually the memory address of the first byte of `my_integer_variable`), assigns it to a pointer-to-`int` variable, and then writes the answer into memory using the indirection operator. Some trickery is needed to force the argument to be treated as a pointer-to-`int` (since this is potentially dangerous and will not compile without some coercion). To do this, `scanf` will use a cast (see Section 1.3.2):

```
int j = 216423324;  /*  some random memory address  */
double *p;          /*  p will point to a double    */

p = j;     /*  won't compile, assigning to *p afterwards
               could be dangerous  */

p = (double*) j; /*   (cast j) will compile, this time compiler assumes
                      you know what you're doing */

*p = 3.1415;   /* will compile, but will probably crash when run, since
                  this memory probably isn't allocated to our program  */
```

Operators on pointers

As well as `&` (address-of) and `*` (indirection), pointers support: assignment from pointers of the same type, all the comparison operators (treating the value of the pointer as a large integer), and also `[]` (indexing) and `+` and `-` (pointer arithmetic).

Indexing a pointer works as follows: suppose `p` is an `int*` (so `p` will be understood as storing the address of an `int`), and imagine that there are several `int`s stored consecutively starting at the address held by `p`. Then `p[i]` (where `i` is an integer variable) is the `i`th `int` starting at the address held by `p` (so `p[0] = *p`). As `i` increases, the address of `p[i]` will increment in jumps of `sizeof(int)` (`sizeof` is described Section 1.9.5). Notice that it is crucial that the compiler knows what kind of pointer `p` is, so it work out how many bytes to add to `p`. There is a very strong connection between pointers used in this ways and arrays, because this is exactly how an array is stored in memory (and how array indexing works):

```
float arr[10];  /*  an array of 10 floats  */
float *fp;      /*  fp will point to floats  */
char  *cp;      /*  cp will point to chars  */

arr[0] = 3.14;  /*  set the zeroth element of arr:
                    (recall that arrays are indexed from 0)  */
fp = &arr[0];   /*  fp points to the zeroth element of arr    */
if (*fp > 3) {
  /*  this is true  */
}

cp = &arr[0];   /*  this will not compile - cp cannot point to floats */
```

```
fp[5] = 12.4    /*  ok  */
if (arr[5] > 12) {
  /*  this is true, fp[5] and arr[5] refer to a double stored
      in the same area of memory  */
}
```

Pointer arithmetic gives another way of stepping though memory. If `p` is a pointer (say a pointer to an `int`), `p+i` is the address of `p[i]`, thus a statement like `p = p+1;` moves `p` forward to point to the next `int`, similarly `p=p-3;` moves `p` back by three. There is more to the connection between arrays and pointers, see below for details.

Pointers-to-`struct`s also accept the `->` operator (see Section 1.4.6).

`const` pointer declarations

You may find functions, such as `printf`, declared with arguments like `const char*` or `char const*` (they mean the same thing). This declaration states that the argument is a pointer-to-a-constant-`char`; the function promises not to change the the memory pointed to by the argument.

Arrays and pointers

As described above, pointers can behave much like arrays. Arrays can, in turn, behave a lot like pointers. Some examples:

```
void f(int* c) {    /*  this function will accept arrays-of-int too  */
  /* .. */
}

int main(void) {
   int *xp;    /*  a pointer-to-int  */
   int x[20];  /*  an array of ints  */

   xp = &x[0];   /*  xp points to x[0]  */
   xp = x;       /*  this is interpreted as the same as the line above  */

   xp = xp + 1;     /*  xp now points to x[1]  */
   xp = &x[0] + 1;  /*  xp will again point to x[1] */
   xp = x + 1;      /*  this is interpreted as the same as the line above  */

   f(xp);     /*  fine  */
   f(&x[0]);  /*  fine, &x[0] is a pointer to x[0] which is an int   */
   f(x);      /*  this is interpreted as the same as the line above  */
}
```

Since a string is an array-of-`char`, there are times when a string literal also behaves like a pointer:

```
/*  You can initialize a pointer-to-char with a string:  */
char *y = "This is a string";

/*  or even an array-of-pointer-to-char from an array of strings  */
char *pets[]  = { "cats", "dogs", "parrots" };

/*  and then use the pointer-to-char as it if were a string:  */
printf("%s\n", pets[2]);
```

The explanation is this: in all but three cases, whenever you use array (e.g. `x`), it will be interpreted as a pointer to the first element of the array (`&x[0]`). The array is said to 'decay' into a pointer to its first element. Thus arrays and pointers behave in the much same way (they can be indexed using `[]`, and accept indirection using `*`). The exceptions are:

```
/*   /examples/chapter1/decayexceptions.c  */
int* xp;    /*  a pointer-to-int  */
int x[20];  /*  an array of ints  */
int t;

/*  1) array initialization from a string literal */
char pet[] = "cat";  /*  the rhs is an array (strings are arrays, so
                         a string literal is too), but we don't assign
                         the address of the letter 'c'  */

/*  2) the address-of operator  */
p = &x;  /*  rhs is interpretated as the address of the first element,
             not the 'address of a pointer-to-first-element'
             which wouldn't make sense unless this pointer were stored
             somewhere, and wouldn't be very useful anyway.   */

/*  3) the sizeof operator  */
t = sizeof(x);  /*  t is set to the size of the whole array x, not
                    sizeof(pointer-to-first-element)
                    which is just the size of a pointer, and thus
                    not as useful.  */
```

Dynamic memory

Pointers are essential if you want to exploit dynamic memory. The function `malloc` can be used to allocate memory at run-time by returning a pointer to the start of a block of memory. The memory can be released when it is no longer needed with `free`. `malloc` returns a generic pointer (a `void*`) which can be converted to any other pointer type automatically.

For every pointer type (for example `int*`) there is a special 'null-pointer', denoted `0` (but converted into a suitable address when compiled) which is guaranteed to be 'invalid' in the sense that no `int` can every be stored with its first byte at `(int*)0`. For this reason, `malloc` returns `(void*)0` when it cannot allocate the memory you ask for. Instead of writing 0, C provides the symbol `NULL` which makes it clearer to the programmer that this is a null pointer, not the integer 0.

1.4.6 Structures

C has a notion similar to Pascal's `record`, called the `struct`, which allows you to keep various related pieces of information together, without having to declare lots of variables:

```
{
  /*  define the type 'struct point' as convenient way of describing
      points in two-dimensions.  This definition can occur at any place
      that a variable declaration would be legal */
  struct point {
    double x;
    double y;
  };

  struct point pt;   /*  declare pt as a 'struct point'  */

  /*  we want to make pt represent (0,1)  */
  pt.x = 0.0;   /*  . gives access to the fields of a struct  */
```

```
    pt.y = 1.0;
  }
```

A quicker way of initializing a structure resembles that for arrays:

```
  struct point the_origin = { 0.0, 0.0 };
```

The type of `pt` is `struct point`, and like any type of known size (the size is known because the compiler knows the size of a `double`), you can declare arrays of them:

```
  struct  point {
    double x;
    double y;
  };

  struct point box[4];

  box[0] = { 0.0, 0.0 };
  box[1] = { 1.0, 0.0 };
  box[2] = { 1.0, 1.1 };
  box[3] = { 0.0, 1.0 };
```

To use a `struct` in more than one part of a program, the definition must occur above the point where it is first used.

```
  struct point {
    double x;
    double y;
  };

  void display(struct point e) {

    printf("x = %f\n", e.x);
    printf("y = %f\n", e.y);
  }
  /* .. */
```

It is very common to pass pointers-to-`struct`s rather than the `struct`s themselves as arguments to a function. If `x` is a pointer-to-`struct`, C has a nice syntax for indirecting the pointer followed by selecting one of the fields, `->`. The function `display` would more normally be written as

```
  /* .. */

  /*  take a pointer-to-struct-point  */
  void display(struct point* e) {

    printf("x = %f\n", e->x);     /*  e->x means (*e).x  */
    printf("y = %f\n", e->y);
  }

  int main(void) {
    /* .. */
    display(&box[0]);
  }
```

`struct`s support assignment from variables of the same type, but do not support comparison operators, not even tests for equality/non-equality.

1.4.7 Enumerated types

C lets you define enumerated types:

```
enum { green, violet, indigo, lilac } colour;  /*  define ‘enum colour’  */
enum colour a_particular_colour = violet;
```

The definition of the type (here `enum colour`) must occur before its first use:

```
enum { green, violet, indigo, lilac } colour;  /*  define enum colour  */
/* .. */

int is_purplish(enum colour c) {
  if (c == violet || c == indigo || c == lilac) {
    return 1;
  } else {
    return 0;
  }
}
```

1.4.8 The `void` type

The keyword `void` is used in two contexts: function declarations and pointers. The pointer type `void*` is a a ‘generic’ pointer type: any pointer can be assigned to a `void*`, and a `void*` can be assigned to any pointer. In function definitions and declarations, it is used to indicate either that the function has no return value, or that it takes no arguments:

```
void f(int i) {
  /*  does something with i but doesn’t return a value.
      (The function must not use the ‘return’ keyword.)  */
}

int random_number(void) {
  /*  takes no arguments but returns a integer  */
}
```

1.4.9 Function pointers (advanced)

Function pointers are a complex and very powerful feature of C. They allow you to pass one function as an argument to another function. We will not describe function pointers here, except for an illustration of how a function can be passed to a CCATSL function, and how to work-out from the declaration of the CCATSL function how your function must be declared:

```
double f(double x) {
  return x*x-1.0;
}

int main(void) {

  double ans, err;

  /*  integrate f over 0, 1 using 256 sub-intervals  */
  ans = RombergCL(f, 0.0, 1.0, &err, 8);
}
```

The CCATSL function `RombergCL` is declared as:

```
double RombergCL(double a, double b, int n, double (*f)(double x),
       double *err);
```

The fourth argument is a pointer to a function which accepts one double argument and returns a double. (Whenever you use the name of a function as an argument to another function, it is treated as a pointer.) Similarly an argument declared as

```
void (*fp)(int, double, const char*)
```

is a pointer to a function which takes an `int`, a `double` and a `char*`, it returns nothing (it returns a `void`), and promises not to modify the memory pointed to by the final argument (the `const` qualifier). Pointers are described in detail in Section 1.4.5.

1.4.10 Type qualifiers (advanced)

Any variable declaration can be prefixed with a *type qualifier*: `const` or `volatile`. We will only describe `const` here and refer the user to a more complete reference for a description of `volatile`.

`const` is used as a 'hint' to the compiler that the variable will not be changed any point within the scope of the declaration. This may allow the compiler to generate code optimized more efficiently, and it will certainly warn you if you modify the variable accidentally.

```
void f(void) {
  const int i=10;  /*  we will not modify i  */
  /* .. */
}
```

A more common use is in conjunction with pointer arguments to a function:

```
void display(const char* message) {  /*  the function guarantees
                                         not to modify message  */
  /* .. */
}
```

1.4.11 Unions and bitfields (advanced)

We will not discuss these.

1.4.12 `typedef` (advanced)

We will not discuss this.

1.5 Associativity and precedence of operators

Table 1.4 gives the rules determining the associativity and precedence of all the operators in C. Associativity means whether an expression like `x R y R z` (where `R` is a operator such as `+` or `<=`) should be evaluated 'left-to-right' i.e. as `(x R y) R z` or 'right-to-left' i.e. as `x R (y R z)`. Precedence determines how an expression like `x R y S z` should be evaluated (now `R` and `S` are *different* operators). If `R` has higher precedence than `S`, it will be evaluated as `(x R y) S z`, while if `S` has higher precedence than `R` it will be treated as `x R (y S z)`.

<table>
<tr><th>Operators in order of precedence</th><th>Associativity</th></tr>
<tr><td>(), [], ->, .</td><td>left to right</td></tr>
<tr><td>!, ~, ++, --, - (unary), * (indirection), & (address-of), sizeof, casts</td><td>right to left</td></tr>
<tr><td>* (multiplication), /, %</td><td>left to right</td></tr>
<tr><td>+, - (subtraction)</td><td>left to right</td></tr>
<tr><td><<, >></td><td>left to right</td></tr>
<tr><td><, <=, >=, ></td><td>left to right</td></tr>
<tr><td>==, !=</td><td>left to right</td></tr>
<tr><td>& (bitwise and)</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>||</td><td>left to right</td></tr>
<tr><td>?:</td><td>right to left</td></tr>
<tr><td>=, +=, -= etc.</td><td>right to left</td></tr>
<tr><td>,</td><td>left to right</td></tr>
</table>

Table 1.4: The associativity and precedence of the operators in C.

1.6 Control Structures

C has the control structures found in most other languages. All involve the use of logical expression (e.g. `a < b`) to affect the flow of execution, so it is important to recall that any expression `e` will be automatically converted into a logical expression if necessary, by interpreting it as `e != 0`.

1.6.1 `if`

The `if` statement has a simple syntax:

```
if (a < b) {
  /*  things to if a < b  */
}
/* .. */
if (a < b) {
  /*  things to do if a < b  */
} else {
  /*  things to do if a >= b  */
}
```

If the compound-statements only contain a single command, the curly brackets can be omitted. For example:

```
if (a < b) a = b;
/* .. */
if (a < b) a = b;
else a = a+1;
```

1.6.2 `while`

C's `while` is almost identical to Pascal's:

```
while (i < 10) {
  /*  things to repeat until i >= 10  */
}
```

1.6.3 `do`

`do` is similar to `while` except that the continuation condition is tested at the end of the loop rather than at the beginning; it follows that the body of the loop is certain to be executed at least once:

```
int a[10];
int i;
/* .. */
i=0;
do {  /*  write out the initial increasing segment of a  */
   printf("%d\n", a[i]);
   i = i+1;
} while (i<9 && a[i] >= a[i-1])
```

1.6.4 `for`

The C `for` statement is a very general looping construction, and has been copied by many other programming languages. A simple use might look like

```
for (i=0; i<10; i=i+1) {
  /*  body of the loop goes here  */
}
```

The parentheses following the `for` keyword contain three expressions, of which the second is interpreted as a logical expression. The first expression (the initialization statement) is evaluated at the start, then the second expression (the continuation condition) is evaluated. If it is true, the body of the loop is executed, then the third expression (the increment statement) is evaluated. The continuation condition is checked again, and if still true, the body of the loop is executed once more. This process continues until the continuation condition fails or a `break` statement is met (see Section 1.6.6). The `for` loop above is equivalent to

```
i = 0;
while (i < 10) {
  /*  body of the loop  */
  i = i+1;
}
```

The three expressions can be arbitrary, even empty. (If the continuation condition is empty, it will be interpreted as true, so the loop will run forever, unless a `break` statement is met.)

```
/*  write out the part of the string a, up to but excluding
    the first occurrence of the character 'R'  */
for (i=0; a[i] != 'R'; i=i+1) {
  printf("%c", a[i]);
}
```

1.6.5 `switch`

`switch` is C's equivalent of Pascal's `case` construction.

```
int option;
/* .. */
switch (option) {
case 1:
  printf("option 1 selected\n");
  /* .. */
  break;
case 2:
  printf("option 2 selected\n");
  break;
default:
  printf("unknown option selected\n");
  break;
}
```

Note that `option` must be a numerical type, and that a `break` statement must be used if you do not want control to 'fall through' to the following cases:

```
int option;
/* .. */
switch (option) {
case 1:
  printf("option 1 selected\n");
case 2:
  printf("option 2 selected\n");
/* .. */
}
```

If `option` is one, both messages will be displayed.

1.6.6 The `break`, `continue` and `goto` statements

These three statements are used to interrupt the flow of control inside loops. `break` passes control to the next statement immediately after the end of the innermost loop:

```
while ( /* .. */ ) {
  while ( /* .. */ ) {
    /* .. */
   break;
  }
  /*  when the break executes, the program continues to execute from here  */
}
```

`break` statements are commonly used with `switch` statements (see above).

`continue` is used in `do`, `while` and `for` loops, to transfer execution to the bottom of the 'body' of the loop. In the case of `do` and `while`, the next thing to happen is the testing of the continuation condition. With a `for` statement, execution continues with the increment statement, then the continuation condition is tested.

`goto` makes the program jump to a given *label*;

```
{
  goto here;
  /* .. */
```

```
    here:     /*  define the label 'here'  */
   /* .. */
  }
```

1.7 Shorthands

C offers the programmer shortened versions of common statements:

```
a += b;  /* equivalent to a = a + b  */
a -= b;  /* equivalent to a = a - b  */
a *= b;  /* equivalent to a = a * b  */
a /= b;  /* equivalent to a = a / b  */
a &= b;  /* equivalent to a = a & b  */
a |= b;  /* equivalent to a = a | b  */
a ^= b;  /* equivalent to a = a ^ b  */
```

A useful syntax for small `if` constructions is the expression

```
b ? c : d  /*  evaluates to c if b is true, and d otherwise  */
```

Another very common syntax is the expression `i++`. This evaluates to `i`, but then increments `i` by 1, and `++i` which first increments `i`, and then returns the new value. `i` can be decremented using the expressions `i--` and `--i`, which behave analogously.

1.8 Declaring and using functions

1.8.1 Introduction

Function definitions in C have the following form:

```
int my_min(int n1, int n2, int n3) {  /*  return the minimum  */

  if (n1 < n2 && n1 < n3) {  /*  is n1 the min?  */
    return n1;
  }
  if (n2 < n3) {  /*  one of n2 and n3 must be the min  */
    return n2;
  }
  return n3;  /*  n3 must be the min  */
}
```

This defines a function `my_min` which calculates the minimum of three integers. The first line starts with the type of the return value (here an `int`), followed by the function's name and the list of *formal parameters*. The body of the function is enclosed in curly brackets. When the body of the function executes, as soon as a `return` statement is found, the expression following the `return` keyword is evaluated and returned as the value of the function (note that you can use `return` more than once).

The formal parameter list is a comma-separated list identifying the types of the function's parameters, and giving them each a name so that they can be referred to in the body of the function.

With `my_min` declared as above, we could call it from any point in our program below the declaration:

```
int main(void) {

  int a=1;
  int b=2;
  int c=3;
  int d;

  /*  set d to the min of a, b and c  */
  d = my_min(a, b, c);
}
```

Local variables

You can declare variables at the start of the function body:

```
/*  return the harmonic mean of three numbers */
float my_mean(float a, float b, float c) {

  float t;

  t = 1.0/a;
  t = t + 1.0/b;
  t = t + 1.0/c;

  return 3.0/t;
}
```

The variable `t` declared here is only visible in the body of the function; any other variable called `t` declared elsewhere will not be effected by calls to `my_mean`:

```
float t;   /*  this is visible everywhere (a ‘global’ variable)  */

float my_mean(float a, float b, float c) {
  float t;
  /*  in the body of the function, references to t effect
      the local variable t, declared on the line above  */
  /* .. */
}

int main(void) {
  float p;
  t = 1;
  p = my_mean(3.0, 4.0, 5.0);
  /*  t still equals one  */
}
```

Each time the function is called, memory for the local variable `t` will be allocated afresh, so its value will not be preserved between calls. If you do want the value to be preserved, use the `static` keyword (see Section 1.3.3).

1.8.2 Pass by value and pass by reference

With the exception of arrays and functions (see below), C always passes arguments ‘by value’: a copy of the value of each argument is passed to the function; the function cannot modify the actual argument passed to it:

```
void foo(int j) {
  j = 0;  /*  modifies the copy of the argument received by the function  */
}

int main(void) {
  int k=10;
  foo(k);
  /*  k still equals 10  */
}
```

If you do want a function to modify its argument you can obtain the desired effect using pointer arguments (see Section 1.4.5) instead:

```
void foo(int *j) {
  *j = 0;
}

int main(void) {
  int k=10;
  foo(&k);
  /*  k now equals 0  */
}
```

This is sometimes known as ‘pass by reference’ in other languages.

1.8.3 Calling functions defined anywhere in your program: prototypes

You should only call a function after it has been declared, otherwise parameter type checking is not performed. However you do not have to *define* a function when you *declare* it:

```
float my_mean(float, float, float);  /*  declares a function my_mean  */
```

You can call `my_mean` from any point below the declaration, because the compiler can generate code for a call to `my_mean` from just the information provided here. The function can now be defined anywhere, even at the bottom of the program, (or maybe in some external library).

1.8.4 Passing arrays, `struct`s and functions as arguments to a function

You can pass an array or `struct` as an argument to a function in the same way as any other type:

```
int zero_array(double a[10]) {
  int i;
  for (i=0; i<10; i=i+1) {
    a[i] = 0.0;
  }
}

int main(void) {
  double a[10];
  zero_array(a);
}
```

Though `struct`s are passed by value (see above for an explanation of ‘by value’), C’s handling of arrays (see in Section 1.4.5) means that arrays are effectively passed ‘by reference’.

If you do not know the size of the array at compile-time, you can declare the function as:

```
int zero_array(double a[]) {
 /* .. */
}
```

but you will probably need to know the size of the array at runtime, and you are likely to end up with something like

```
int zero_array(double a[], int size_of_array) {
 /* .. */
}
```

You can also pass functions as arguments to a function, using function pointers (see Section 1.4.9).

1.9 Some hilights of the standard C library

This section describes the most commonly used functions in the 'standard C library'. In C, a function should be declared before it can be used (see Section 1.8.3). Declarations of the functions in the standard library are provided by a number of header files, which can be read-in automatically at the start of your program. For example, to read in the header file `stdio.h`, which declares the input-output functions, put

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

near the top of your program.

1.9.1 Mathematical functions

`abs`

Returns the absolute value of an integer. (Declared in `stdlib.h`.) Use `fabs` for floating-point arguments.

```
int abs(int i);
```

`fabs`

Returns the absolute value of a floating-point number. (Declared in `math.h`.) Use `abs` for integer arguments.

```
double fabs(double x);
```

`floor`, `ceil`, `rint`

These functions return respectively the greatest integer not exceeding, the least integer not less than, and the closest integer to, their argument, returning the answer as a floating-point number. They are all declared in `math.h`, and have very similar declarations, e.g.

```
double floor(double x);
```

cos, sin, tan, acos, asin, atan, cosh, sinh, tanh, acosh, asinh, atanh, exp, log, sqrt, log10

These functions compute various common mathematical functions. Note that `log` returns natural logs, while `log10` computes logs to base 10. They are all declared in `math.h`, and have very similar declarations, e.g.

```
double cos(double x);
```

A typical use might look like

```
double a;
double b;

b = 0.5;
a = cos(b);
```

atan2

This function computes the arctangent in $(-\pi, \pi]$ of the ratio x/y, (it works in the case $y = 0$). (Declared in `stdlib.h`.)

```
double atan2(double x, double y);
```

pow

`pow` returns x raised to the power of y. If x is negative, then y must be integral. (Declared in `math.h`.)

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

drand48, srand48

These functions are the recommended way of generating uniform pseudo-random numbers. Successive calls to `drand48` return numbers in the range $[0, 1)$. The sequence will be the same each time the program is run unless the *seed* to the random number generator is changed. `srand48` can be used to set the seed to a specific number. (Though not part of standard C, you may find that both are declared in `stdlib.h`. If your compiler complains that they are 'implicitly declared', you will have to declare them yourself.)

```
double drand48(void);
void srand48(long int);
```

To ensure that your program uses a different sequence of random numbers each time the program is run, use

```
#include <time.h>
/* .. */
srand48((long)time(NULL));   /*  time is declared in time.h  */
```

To obtain a uniform random integer in the the set $\{1, \ldots, n\}$ use

```
int i;
/* .. */
i = 1+(int)(n*drand48());
```

isnan, isinf, finite

These (non-standard) functions allow you to detect the special floating-point entities NaN 'not-a-number' +Infinity and -Infinity. (Declared in math.h.)

```
int isnan(double x);
int isinf(double x);
int finite(double x);
```

isnan returns non-zero if its argument is represents 'not-a-number' (NaN) and zero otherwise. isinf returns -1 if its argument represents -Infinity, $+1$ if the argument represents +Infinity and zero otherwise. finite returns non-zero if the argument does not represent +Infinity, -Infinity or NaN.

1.9.2 String functions

Recall that strings in C are just arrays-of-char in which the final character has numerical value 0. The standard library provides a number of useful routines for manipulating strings.

strlen

strlen returns the number of characters in a string, not including the zero-terminator. (Declared in string.h)

```
size_t strlen(const char* s);  /*  size_t is a predefined integer type  */
```

strcmp

strcmp compares two strings lexicographically. It returns a negative number if the first string is less than the second, zero if the two strings are identical and a positive number if the first string is greater than the second. (Declared in string.h.)

```
int strcmp(const char *s1, const char *s2);
```

strcpy and strncpy

strcpy copies one string into another. The destination string must have enough room for all the characters of the source string, including room for the zero-terminator. The function returns a pointer to the start of the destination string. If available you should use strncpy instead which allows you to specify the amount of space in the destination string. (Declared in string.h.)

```
char *strcpy(char *destination_str, const char *source_str);
 /*  size_t is a predefined integer type  */
char *strncpy(char *destination_str, const char *source_str, size_t size);
```

atof, atoi

atof and atoi convert a string to a floating-point number and an integer respectively. (Declared in stdlib.h.)

```
double atof(const char *str);
int atof(const char *str);
```

1.9.3 Reading and Writing: the screen and keyboard

The C library provides three *streams*: `stdin`, `stdout` and `stderr` for input and output. `stdin` is an input stream, used to receive input from the user. `stdout` is the normal output stream, and `stderr` is a second output stream, traditionally used to report error messages (or other unexpected output).

Usually `stdin` collects characters typed at the keyboard, while characters sent to `stdout` are echoed to the screen. On some systems, `stdin` and `stdout` can be made to read to and write from files: running the command `my_prog <input.txt` runs `my_prog`, but empties the contents of the file `input.txt` into `stdin` rather than waiting for keyboard input. Such systems usually also let you redirect `stdout` and `stderr`, which is useful if you want to save or printout your program's output: `my_prog <input.txt >output.txt 2>errors.txt` would redirect all three streams to appropriate files.

It is possible that output to the screen produced by one line does not appear before the next line of the program executes. If this is going to be a problem, (for example if you're writing out a prompt and then reading input from the user, you need the prompt to appear before waiting for input) use `fflush` (see below).

printf

`printf` is the usual method for writing information to the screen (similar to Pascal's `Write` and `Writeln`). (Declared in `stdio.h`.)

A simple use of `printf` might look like

```
printf("Some message.");   /*  argument is a string  */
```

This copies its argument to the standard output stream, `stdout` (usually characters sent to `stdout` will be sent to the screen). To write the value of a variable, use a `%` sign followed by a letter indicating the type of the variable

```
int i;
/* .. */
printf("Iteration %d\n", i);  /*  \n  is the newline character
                             (see section on the char type for more info)  */
```

To write the value of variables of other types, use `%f` for `double`s and `float`s, `%c` for `char`s, and `%s` for strings. (There are conversions for other types, and for each type you can request that the variable be displayed in a number of different formats, but we will not describe them all here.)

You can write the value of several variables at once:

```
int i;
char names[2][80] = { "test", "main run" };
double val;
int iter;
/* .. */
printf("Problem %s: iteration %d, value=%f\n", names[i], iter, val);
```

or specify a field with and precision (see a reference manual for more detailed information):

```
int iter;
double val;
/* .. */
printf("Iteration %10i", iter);  /*  use a field-width of 10 characters  */
printf("value %10.5f\n", val);  /*  field-width of 10,  5 decimal places  */
```

scanf

`scanf` reads and interprets characters from the standard input stream `stdin`, which usually receives what the user types at the keyboard. (Declared in `stdio.h`.)

Simple uses of `scanf` might look like

```
int i, j, k;
/* .. */
scanf("%d", &i);  /*  read an integer into i  */
/* .. */
scanf("%d %d %d", &i, &j, &k);  /*  read in 3 integers,
                                     separated by whitespace */
```

To read in other types, use `%f` for `float`s, `%lf` for `double`s (it is a common error to use `%f` for `double`s—unlike `printf`, `scanf` will read the wrong number), and `%s` for strings. When reading in strings, the input stream will be split up into 'words' (sequences of non-whitespace characters), and each one will be stored in a separate string argument.

```
double r;
char word1[100],word2[100];
/* .. */
scanf("%lf", &r);  /*  read in a double  */
scanf("%s %s", word1, word2);  /*  read in two words (both must be
                                   shorter than 100 characters  */
```

fflush

`fflush` flushes a stream (waits until all pending output is written). (Declared in `stdio.h`.) The most common use is:

```
double sigma;
/* .. */
printf("Enter the value for sigma: ");
fflush(stdout);  /*  wait for the prompt to be written on the screen  */
scanf("%lf",&sigma);
```

`fflush` can be used with any stream open for writing.

1.9.4 Reading and Writing: disk files

Writing and reading to disk files is accomplished in much the same way as reading and writing to the screen. The approach taken in C is to associate a stream with the file using `fopen`, and then use `fprintf` and `fscanf` for reading and writing. These behave exactly like `printf` and `scanf` except that they allow you to specify which stream they operate on (`printf` always uses `stdout`, while `scanf` uses `stdin`). When you have finished using the stream you should close it with `fclose`.

fopen

`fopen` opens a file and associates a stream with it. (Declared in `stdio.h`.)

```
FILE *fopen(char *path, char *mode);
```

`path` is a string specifying the name of the file, while `mode` is a string indicating how the file is to be opened, typically either `"r"` to read from the file or `"w"` to write to it. If the file cannot be opened for some reason, `fopen` returns `NULL`.

```
FILE *my_file;
/* .. */
my_file = fopen("datafile.txt", "r");  /*  open datafile.txt for reading  */
if (my_file == NULL) {
  fprintf(stderr, "Can’t open datafile.txt for reading\n");
  exit(1);
}
```

The stream may be access via `fprintf` or `fscanf`, and should be subsequently closed with `fclose`. (In windows programs, `HaltCL` should be used instead of `exit`.)

fclose

`fclose` closes a stream previously opened with `fopen`. (Declared in `stdio.h`.)

```
int fclose(FILE *stream);
```

A simple use of `fclose` might look like

```
FILE *my_file;
/* .. */
my_file = fopen("datafile.txt", "r");
/* .. */
fclose(my_file);
```

fprintf

`fprintf` is the analogue of `printf` for use with arbitrary streams. (Declared in `stdio.h`.)

```
int fprintf(FILE *stream, const char *format, ... );
```

For example, to open a disk-file and write some text to it:

```
FILE *my_file;
int i;
/* .. */
my_file=fopen("datafile.txt","w");
fprintf(my_file, "i=%d", i);
/* .. */
fclose(my_file);
```

The meaning of the second and subsequent arguments to `fprintf` are described in `printf`.

fscanf

`fscanf` is the analogue of `scanf` for use with arbitrary streams. (Declared in `stdio.h`.)

```
int fscanf(FILE *stream, const char *format, ... );
```

For example, to open a disk-file and read some text from it:

```
FILE *my_file;
int i;
/* .. */
my_file=fopen("datafile.txt","r");
fscanf(my_file, "%d", &i);  /*  read an integer from the file into i  */
/* .. */
fclose(my_file);
```

For a description of the meaning of the second and subsequent arguments, see `scanf`.

1.9.5 Miscellaneous

exit

`exit` causes immediate termination of the program. (Declared in `stdlib.h`.) (In windows programs, `HaltCL` should be used instead.)

```
void exit(int status);
```

time

(Non-standard) `time` returns the number of seconds elapsed since 00:00:00 GMT, January 1, 1970. (Declared in `time.h`.)

```
time_t time(time_t *tp);  /*  time_t is a pre-defined integer type  */
```

If `tp` is not `NULL`, the number of seconds is also stored in `tp`. For example:

```
printf("There have elapsed %li seconds since 00:00:00 GMT, January 1\n",
       (long)time(NULL));  /*  assumes time_t is a long int  */
```

Another common use is to seed the random number generator:

```
srand48((long)time(NULL));  /*  time is declared in time.h  */
```

argc and argv - accessing command line arguments

We will not discuss this here.

malloc

`malloc` is used to request blocks of memory at runtime. (Declared in `stdlib.h`.)

```
void *malloc(size_t size);  /*  size_t is a predefined integer type  */
```

The return value is a pointer to the start of a contiguous block of memory of `size` bytes. When the memory has been finished with, you can release it by calling `free`. If there is not enough memory available to satisfy the request, `malloc` returns `NULL`.

free

`free` releases memory previously allocated by `malloc`. (Declared in `stdlib.h`.)

```
void free(void *p);
```

sizeof

`sizeof` returns the number of bytes occupied by a variable or type (it is not actually part of the standard C library but part of the C language, so you do not need to `#include` a header file to use it). For example:

```
int i;
/* .. */
printf("ints occupy \%d bytes\n",sizeof(i)");  /*  sizeof(int) works too  */
```

1.10 Preprocessor directives

One of C's most useful features is its preprocessor. This is a part of compilation which occurs before the main compilation, and performs simple tasks such as including header files. We will only describe two of its features.

`#include`

The `#include` preprocessor command inserts a specified file into the file being compiled. It is often used to include header files containing many function declarations:

```
#include <catam.h> /* access the CCATSL library */
```

this in turn contains

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

which declares the standard IO functions (such as `printf`).

`#define`

The simplest use of the `#define` pre-processor command looks like:

```
#define MAX_ITERS 1000
/* .. */
int i;
/* .. */
for (i=1; i<=MAX_ITERS; i++) {
  /* .. */
}
```

This asks the preprocessor to replace occurrences of the word `MAX_ITERS` with `1000` (this is a common way to define constants in C). `#define` can also be used to define *macros* which behave like functions, but we will not discuss these.

Chapter 2

Mathematical functions

2.1 Ordinary Differential Equations

The CCATSL library provides several routines for solving ODEs of first order, $\dot{x} = a(x, t)$, and also for solving the more general system of ODEs of the form:

$$\begin{aligned} \dot{x}_1 &= a_1(t, x_1, \ldots, x_n) \\ &\vdots \\ \dot{x}_n &= a_n(t, x_1, \ldots, x_n). \end{aligned} \tag{2.1}$$

To handle a second order ODE (or a more complex system), you must first re-write the problem in the form above. For example, suppose you want to solve

$$\ddot{x} = t\dot{x} - 3t - x^2 + 1,$$

subject to $x(t_0) = \alpha$, $\dot{x}(t_0) = \beta$. By introducing the variable $z(t)$, defined by $z(t) = \dot{x}(t)$, we can find the solution to the ODE above by finding the x-part of the solution to

$$\begin{aligned} \dot{x} &= z \\ \dot{z} &= tz - 3t - x^2 + 1 \end{aligned}$$

subject to $x(t_0) = \alpha$, $z(t_0) = \beta$.

There are three CCATSL routines for ODE solving. The simplest, `Rk4CL`, is fast and easy to use, but employs a fixed stepsize and gives no indication of the error in the solution. The second, `RkfCL`, is more sophisticated; it adjusts the stepsize when necessary and allows control over the error introduced at each step. The final routine, `Rkf45CL`, also adjusts the stepsize automatically, but allows control of the error over the whole integration interval.

Fourth order Runge-Kutta, `Rk4CL`

This is the simplest method provided by CCATSL to solve ODEs and assumes that problem has been written in the form of (2.1). The CCATSL routine implementing Fourth order Runge-Kutta is called `Rk4CL`, and is declared as follows:

```
void Rk4CL (int n,
            double *ti,
            double dt,
            ODEFunctionCT F,
            double *x,
            double *xdot);
```

The arguments have the following meanings:

`n` The size of the system.

`ti` Pointer to a variable holding the initial time. When `Rk4CL` returns, it will have been updated to the time at the end of the integration interval.

`dt` The stepsize.

`F` A user-defined function which implements the system of ODEs above by computing $\dot{x}_1, \ldots, \dot{x}_n$ from the values of t and $x_1, \ldots, x_n$). If something goes wrong, the function should set `ErrorFlagCD=true` before returning, which will cause `Rk4CL` to abort. (See below for an illustration.)

`x` Array used to hold the values of $x_1, \ldots, x_n$. Before calling `Rk4CL` for the first time, these should be set to the initial conditions. When `Rk4CL` returns, they will have been updated to hold the values of $x_1, \ldots, x_n$ at the end of the integration interval.

`xdot` Array which will be used by the user-defined function `F` to pass back the values of $\dot{x}_1, \ldots, \dot{x}_n$ to the library.

(If you're confused by pointers, arrays, or arguments declared as `double *`*, don't worry. You can find the details in Section 1.4.5, but they are easier to use than to explain. If a routine asks for a 'pointer to a* `double`*', you just setup a variable of type* `double` *as normal, and then put a* `&` *symbol in front of the variable name. The following example should help to clarify this.)*

Here's a simple example using `Rk4CL` to solve an ODE and `XYCurveCL` (Section 3.1) to plot the solution:

```
/*   /examples/chapter2/rk4demo.c   */
#include <catam.h>

void a(double t, double *x, double *xdot)
{
  xdot[0]=x[1];                     /* x-dot = z              */
  xdot[1]=t*x[1]-3*t-x[0]*x[0]+1;  /* z-dot = tz -3t -x^2 +1 */
}

int MainCL(void)
{
  double dt=0.001; /* integration stepsize */
  double x[2];
  double xp[2];
  double xdata[1000];
  double tdata[1000];
  double t;
  int i;
  t=0.0;                        /* initial time is t=0 */
  x[0]=0.0;                     /* initial condition x=0 */
  x[1]=0.0;                     /* initial condition z=0 */
  for (i=0; i<1000; i++) {
    Rk4CL(2,&t,dt,a,x,xp);      /* perform a single Rk4 step */
    xdata[i]=x[0];              /* store x */
    tdata[i]=t;                 /* store t */
  }
  /* plot the solution */
  XYCurveCL(tdata,xdata,1000,1,JOIN,RedCC,AUTOAXES);
  return 0;
}
```

Another way of writing this program, which avoids storing the values of `x` and `t`, is to use

XYDrawCL (see Section 3.4).

Runge-Kutta-Fehlberg, RkfCL

This is a more powerful technique for solving ODEs than fourth-order Runge-Kutta. It automatically changes the stepsize when necessary and allows control of the local error (the error introduced at each timestep).

```
boolean RKfCL (int n,
               double aberr,
               double relerr,
               double *t,
               double *dt,
               double dtmin,
               ODEFunctionCT F,
               double *x,
               double *xdot,
               int *nleft);
```

- `n` The size of the system.
- `aberr` The acceptable absolute error over each integration step.
- `relerr` The acceptable relative error at each step.
- `t` Pointer to a variable initially set by the user to be the time at the start of the integration interval. When `RkfCL` returns, it will have been incremented to reflect the time at the end of the integration interval.
- `dt` Pointer to a variable holding the desired step length. This value may be changed if `RkfCL` finds that a smaller stepsize is necessary to achieve the required tolerances. The value held by `dt` when `RKfCL` returns may differ from the length of the integration interval.
- `dtmin` The minimum stepsize. If `RkfCL` cannot meet the required tolerances without using a stepsize below than this, `RkfCL` aborts.
- `F` A user-defined function which implements the system of ODEs above by computing $\dot{x}_1, \ldots, \dot{x}_n$ from the values of t and $x_1, \ldots, x_n$). If something goes wrong, the function should set `ErrorFlagCD=true` before returning, which will cause `RkfCL` to abort. (See below for an illustration.)
- `x` Array used to hold the values of $x_1, \ldots, x_n$. Before calling `RkfCL` for the first time, these should be set to the initial conditions. When `RkfCL` returns, they will have been updated to hold the values of $x_1, \ldots, x_n$ at the end of the integration interval.
- `xdot` Array used by the user-defined function `F` to pass back the values of $\dot{x}_1, \ldots, \dot{x}_n$ to the library.
- `nleft` Pointer to a variable used to indicate how many integration steps remain until some desired time is reached, based on the current `dt`. Usually you want to integrate up to some fixed time T; you would choose N to be the desired number of integration steps, passing this via `nleft` and set the suggested step size to be T/N. `RkfCL` will then adjust N appropriately whenever an integration step is completed, or when it changes `dt`.

The return value is `true` if the integration tolerances can be met and `false` if they cannot.

(If you're confused by pointers, arrays, or arguments declared as `double *`, don't worry. You can find the details in Section 1.4.5, but they are easier to use than to explain. If a routine asks for a 'pointer to a `double`', you just setup a variable of type `double` as normal, and then put a `&` symbol in front of the variable name. The following example should help to clarify this.)

Here is an example of how to use `RkfCL` to solve the ODE of the Van der Pol oscillator:

$$\ddot{x} + \mu\dot{x}(x^2 - 1) + x = 0.$$

```
/*   /examples/chapter2/rkfdemo.c  */
#include <catam.h>
double mu=5.0;

void a(double t, double *x, double *xdot)
{
  xdot[0]=x[1];
  xdot[1]=-x[0]-mu*x[1]*(x[0]*x[0]-1);
}

int MainCL(void)
{
  double aberr=1e-5;
  double relerr=1e-5;
  double dtmin=0.001;
  double tfinal=25.0;
  double x[2];         /* will store x and z         */
  double xp[2];        /* will store x-dot and z-dot */
  double t;
  double xdata[1000];
  double tdata[1000];
  int nres;            /* number of results stored */
  double dt;           /* integration stepsize */
  int nleft;           /* number of steps left */

  t=0.0;              /* setup initial conditions */
  x[0]=-1.42;
  x[1]=0.26;
  dt=0.2;
  nleft=(int)(tfinal/dt); /* workout a reasonable stepsize */
  dt=tfinal/nleft;
  nres=0;
  while (nleft>0) {
    if (!RkfCL(2,aberr,relerr,&t,&dt,dtmin,a,x,xp,&nleft)) {
      printf("tolerances can't be met, sorry\n");
      HaltCL();
    }
    if (nres<1000) {     /* careful not to overflow arrays */
      xdata[nres]=x[0];
      tdata[nres]=t;
      nres=nres+1;
    }
  }
  /* plot the solution */
  XYCurveCL(tdata,xdata,nres,1,JOIN,RedCC,AUTOAXES);
  return 0;
}
```

Rkf45CL

This is the most advanced ODE solving routine in CCATSL. It automatically chooses, and if necessary varies, the integration timestep and allows for error control over the whole integration interval.

```
void Rkf45CL (int n,
              double aberr,
              double relerr,
              double tend,
              double *t,
              ODEFunction F,
              double *x,
              double *xdot,
              int *control);
```

`n` The size of the system.

`aberr` The absolute error tolerance. You can set `aberr` to zero and `Rkf45CL` will report back if this leads to a problem

`relerr` The relative error tolerance. `relerr` should be more than about 1e-13 but you can try smaller values.

`tend` The time at the end of the integration interval.

`t` Pointer to a variable holding the time at the start of the integration interval. When `Rkf45CL` returns, this will have been updated appropriately.

`F` A user-defined function which implements the system of ODEs above by computing $\dot{x}_1, \ldots, \dot{x}_n$ from the values of t and $x_1, \ldots, x_n$). If something goes wrong, the function should set `ErrorFlagCD=true` before returning, which will cause `Rkf45CL` to abort. (See below for an illustration.)

`x` Array used to hold the values of $x_1, \ldots, x_n$. Before calling `Rkf45CL` for the first time, these should be set to the initial conditions. When `Rkf45CL` returns, they will have been updated to hold the values of $x_1, \ldots, x_n$ at the end of the integration interval.

`xdot` Array used by the user-defined function `F` to pass back the values of $\dot{x}_1, \ldots, \dot{x}_n$ to the library.

`control` See below.

The parameter `control` is a pointer to a variable used to control the behaviour of `Rkf45CL`. The first call to `Rkf45CL` should set the variable to `1` or `-1`. In the first case, `Rkf45CL` will try to integrate up to time `tend` in one go, meeting the required tolerances. The second case causes `Rkf45CL` to return after each substep, with `t` and the array `x` set accordingly. When `Rkf45CL` returns, the variable acts as a status indicator. Values of `2` and `-2` correspond to successful integrations in the two cases just described (in the step-by-step mode, you should leave `control` at `-2` when you next call `Rkf45CL`). Other possible values are:

3 `relerr` was too small, but has been changed to a more appropriate value, you may continue if you like.

4 More than 5000 steps have been taken and `tend` has still not been reached, you may continue if you like.

5 Solution has vanished and `aberr=0`, you probably want to set `aberr` to a non-zero value and do a single step here (by setting the variable pointer to by `control` to -2).

6 Unable to achieve the desired tolerances, change `relerr` or `aberr` and try again.

7 `Rkf45CL` thinks it wants a stepsize greater than the whole integration interval, the problem is probably stiff (CCATSL has no suitable routines for such problems).

8 The input settings were invalid.

An example showing the use of `Rkf45CL` is `c01orbit.c`

2.2 Integration

CCATSL has two routines for computing definite integrals

$$I = \int_a^b f(x)\,dx.$$

The simpler of the two, `RombergCL`, uses Romberg extrapolation to calculate an approximation to I and to give an estimate of the error. Since it divides the interval $[a, b]$ into a pre-determined number of equal subintervals, it is likely to perform poorly when the integrand varies rapidly or is singular somewhere inside $[a, b]$. The alternative routine, `QuadCL` uses an *adaptive* method for subdividing the interval and copes much better with these integrands. It can also be required to try to achieve user-specified error tolerances.

Romberg Extrapolation, RombergCL

`RombergCL` tries to calculate a definite integral using Romberg extrapolation with a user-specified number of sub-intervals. This routine is easy to use but is unable to handle badly behaved integrands very accurately. (The routine `QuadCL` is more suitable for these problems.)

```
double RombergCL (double (*f)(double x),
                  double a,
                  double b,
                  double *err,
                  int n);
```

`f`	A user-defined function implementing the integrand.
`a`	The lower endpoint of the integration interval.
`b`	The upper endpoint.
`err`	A pointer to variable which will hold an estimate of the error in the integral when `RombergCL` returns.
`n`	Determines the number of subintervals to use: `RombergCL` will use a total of 2^n subintervals (and hence $2^n + 1$ function evaluations). Try `n=5` to start with; if the error (see `err` below) is too large, increase `n` further (if your function is badly behaved or singular in `[a,b]`, you should consider using the more powerful routine `QuadCL`).

The return value is the estimate of the integral. A typical use of `RombergCL` might look like

```
/*   /examples/chapter2/romb.c   */
#include <catam.h>

double f(double x)
{
  return sqrt(1-exp(-x));
}

int MainCL(void)
{
  double errest;
  double answer;

  answer=RombergCL(f,0.0,1.0,&errest,5);
  printf("Integral is %f\n",answer);
```

```
    return 0;
  }
```

A complete example can be found in `a03romb.c`.

Adaptive Quadrature, `QuadCL`

This routine uses an adaptive quadrature method to evaluate definite integrals, and should cope reasonably well with singular and badly behaved integrands.

```
double QuadCL (double (*f)(double x)
               double a,
               double b,
               double aberr,
               double relerr,
               double *err,
               double *flag);
```

`f`	A user-defined function implementing the integrand.
`a, b`	Respectively the lower and upper endpoints of the integration interval.
`aberr`	The acceptable absolute error.
`relerr`	The acceptable relative error. (`QuadCL` is satisfied if it can meet *either* of the specified error tolerances.)
`err`	Pointer to a variable which will hold an estimate of the actual (absolute) error in the integral when `QuadCL` returns.
`flag`	Pointer to a variable used as a 'reliability indicator'. If, when `QuadCL` returns this variable is 0.0, all is well. Otherwise, the integer part gives the number of subintervals where convergence could not be achieved and the fractional part is the proportion of the interval `[a,b]` still to be integrated over.

The return value is the estimate of the integral.

2.3 Matrix routines

A common matrix task is the solution of a set of simultaneous linear equations, since this can be conveniently expressed as

$$Ax = z$$

where A is an n by n matrix and x and z are vectors of size n. CCATSL provides several routines for this: `GaussElimCL` solves the system above using Gaussian elimination with pivoting, while `DecomposeCL` and `SolveCL` respectively decompose a matrix into a more useful form and use the resulting decomposition to solve the equation. The `DecomposeCL/SolveCL` approach is preferable when you want to solve the equation above for several different z since `DecomposeCL` does not involve z and `SolveCL` is very quick.

Two special cases are where the matrix A is *tridiagonal*

$$A = \begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & & 0 \\ 0 & a_3 & b_3 & c_3 & & \vdots \\ 0 & 0 & a_4 & b_4 & & 0 \\ \vdots & & & & \ddots & c_{n-1} \\ 0 & 0 & \dots & 0 & a_n & b_n \end{pmatrix}$$

for which the routine `TridiagCL` is better suited, or *banded* such as

$$A = \begin{pmatrix} b_1 & c_1 & d_1 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & d_2 & & 0 \\ 0 & a_3 & b_3 & c_3 & & \vdots \\ 0 & 0 & a_4 & b_4 & & d_{n-2} \\ \vdots & & & & \ddots & c_{n-1} \\ 0 & 0 & \dots & 0 & a_n & b_n \end{pmatrix}$$

which are best handled with `BandCL`. The routine `InvertCL` inverts a matrix.

Another common problem is the determination of eigenvalues and eigenvectors. CCATSL has three routines for this: `EigenMaxCL` finds an eigenvalue of greatest modulus and an eigenvector with this eigenvalue, `EigenValCL` takes a parameter λ and finds an eigenvalue θ minimising $|\theta - \lambda|$, together with a corresponding eigenvector. For symmetric A, `JacobiCL` returns all the eigenvalues and eigenvectors, using the method of Jacobi.

If A is an m by n matrix, a singular value decomposition (i.e., a decomposition of the form $A = USV^T$ where U and V are orthogonal and S is diagonal) can be found using the routine `SvdCL`.

Gaussian Elimination with Pivoting, `GaussElimCL`

This routines use Gaussian elimination with pivoting to solve the system of simultaneous linear equations

$$Ax = z$$

where A is an n by n matrix and x and z are vectors of size n.

```
double GaussElimCL (int n,
                    int ncols,
                    double *a,
                    double *z,
                    double *x);
```

- `n` The order of the matrix A.
- `ncols` Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.
- `a` Array holding the matrix A.
- `z` Array holding the vector z.
- `x` Array to receive the solution vector x.

The return value is the determinant of the matrix A. A typical use of `GaussElimCL` might look like

```
/*   /examples/chapter2/elim.c   */
#include <catam.h>
int MainCL(void)
{
  double a[2][2];
  double z[2];
  double x[2];
  double det;
  a[0][0]=1; a[0][1]=1;
  a[1][0]=0; a[1][1]=1;
  z[0]=2;
```

```
    z[1]=3;
    det=GaussElimCL(2,2,a,z,x);
    printf("determinant is %lf\n",det);
    return 0;
}
```

A complete example of how to use `GaussElimCL` can be found in `b09matrx.c`.

DecomposeCL

`DecomposeCL` is designed to be used in combination with `SolveCL` (see below) to solve a set of simultaneous linear equations expressed in the form $Ax = z$. Once the matrix A has been decomposed with `DecomposeCL`, `SolveCL` can be called to solve for a given z. Since `SolveCL` is much faster than `GaussElimCL`, if the same set of equations is to be solved for different vectors z, the `DecomposeCL/SolveCL` approach is to be preferred to `GaussElimCL`.

```
double DecomposeCL (int n,
                    int ncols,
                    double *a);
```

- `n` The order of the matrix A.
- `ncols` Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.
- `a` Array holding the matrix A. When `DecomposeCL` returns, A will have been replaced by its decomposed form. If you want to retain the original matrix A, you must make a copy of it before calling `DecomposeCL`.

The return value is the determinant of the matrix A. See `b09matrx.c` for an example of how to use `DecomposeCL`.

ConditionCL

`ConditionCL` returns the last condition number of the last matrix decomposition (see `DecomposeCL`) or inversion (see `InvertCL`).

SolveCL

This routine complements `DecomposeCL` by taking a matrix in decomposed form and a vector z and solving the system of linear equations $Ax = z$. It is faster than Gaussian elimination (`GaussElimCL`) and is thus more efficient for solving for several different vectors z.

```
void SolveCL (int n,
              int ncols,
              double *a,
              double *z);
```

- `n` The order of the matrix A.
- `ncols` Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.
- `a` The matrix A.
- `z` Array to hold the right-hand side z. When `SolveCL` returns, it will have been overwritten and will hold the solution vector x.

The example program `b09matrx.c` demonstrates the use of `SolveCL`.

Tridiagonal Linear System Solver, `TridiagCL`

This routine solves the linear system $Ax = z$ for the special case where A is tridiagonal

$$A = \begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & & 0 \\ 0 & a_3 & b_3 & c_3 & & \vdots \\ 0 & 0 & a_4 & b_4 & & 0 \\ \vdots & & & & \ddots & c_{n-1} \\ 0 & 0 & \dots & 0 & a_n & b_n \end{pmatrix}.$$

```
void TridiagCL (int n,
                double *a,
                double *b,
                double *c,
                double *z,
                double *x);
```

- `n` The order of the matrix A.
- `a` Array holding the values of $a_1, \dots, a_n$. (The value of a_1 is irrelevant but it must be present.)
- `b` Array holding $b_1, \dots, b_n$.
- `c` Array holding $c_1, \dots, c_n$. (The value of c_n is irrelevant.)
- `z` Array holding the vector z.
- `x` Array to hold the solution vector x.

A typical use of `TridiagCL` is the following:

```
/*   /example/chapter2/tridiag.c  */
#include <catam.h>
int MainCL(void)
{
  double a[5],b[5],c[5],z[5],x[5];
  c[0]= 1; c[1]= 1; c[2]= 1; c[3]=1;
  b[0]=-2; b[1]=-2; b[2]=-2; b[3]=-2; b[4]=-2;
  a[1]= 1; a[2]= 1; a[3]= 1; a[4]= 1;
  z[0]=0.5; z[1]=0.5; z[2]=0.5; z[3]=0.5; z[4]=0.5;
  TridiagCL(5,a,b,c,z,x);
}
```

Banded Linear System Solver, `BandCL`

This routine solves the linear system $Ax = z$ when A is banded: $A_{ij} = 0$ if $i - j > l$ or $i - j < -u$ where $0 \le l, u \le n$. The numbers l and u are called the *lower bandwidth* and the *upper bandwidth* respectively. For example, the matrix

$$A = \begin{pmatrix} b_1 & c_1 & d_1 & 0 & 0 \\ a_2 & b_2 & c_2 & d_2 & 0 \\ 0 & a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & a_4 & b_4 & c_4 \\ 0 & 0 & 0 & a_5 & b_5 \end{pmatrix}$$

has a lower bandwidth of 1 and an upper bandwidth of 2. The advantage of `BandCL` over a standard Gaussian elimination routine such as `GaussElimCL` is that it exploits the sparse structure to speed up the computation, and that it can handle much larger matrices, since the full matrix A is never actually stored. Instead a *compact form* of A must be setup and passed to `BandCL`. The compact form of the matrix above is

$$\begin{pmatrix} \cdot & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & \cdot \\ a_5 & b_5 & \cdot & \cdot \end{pmatrix}$$

where dotted entries are irrelevant and ignored by `BandCL`. Note that if A is n by n then the compact form of A is n by $(1 + l + u)$ where l and u are the lower and upper bandwidths.

```
double BandCL (int n,
               int ncols,
               int lbw,
               int ubw,
               double *cpact,
               double *z,
               double *x);
```

`n`	The order of the matrix A.
`ncols`	Normally `ncols=1+lbw+ubw`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `cpact`.
`lbw`	The lower bandwidth of the matrix.
`ubw`	The upper bandwidth.
`cpact`	Array holding the matrix A stored in compact form. (See above for details of the compact form.)
`z`	Array holding the vector z.
`x`	Array to hold the solution vector x.

The return value is the determinant of the matrix. `b09matrx.c` contains an example of how to use `BandCL`.

Matrix Inversion, `InvertCL`

`InvertCL` inverts a square matrix and returns its determinant. The inverse overwrites the original matrix; thus if you want to retain the original matrix you must make a copy of it prior to calling `InvertCL`.

```
double InvertCL (int n,
                 int ncols,
                 void *a);
```

`n`	The order of the matrix.
`ncols`	Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.
`a`	Array holding the matrix A. When `InvertCL` returns, it will have been overwritten with the inverse matrix.

The return value is the determinant of the matrix A. `b09matrx.c` has an example of how to use `InvertCL`.

Singular Value Decomposition, `SvdCL`

The CCATSL routine `SvdCL` computes a decomposition of the form $A = USV^T$ where U and V are orthogonal and S is diagonal. If A is m by n, then U is m by m, S is m by n and V is n by n, thus if $m > n$, the last $m - n$ rows of S will be zero and the last $m - n$ columns of U will not have any special significance. Similarly if $n > m$, the last $n - m$ columns of S will be zero and the last $n - m$ columns of V are not significant. Since S is diagonal, `SvdCL` just returns the diagonal elements (the *singular values*) of S as a one-dimensional vector.

```
int SvdCL (int m,
           int n,
           int ncols,
           double *a,
           double *u,
           double *v,
           double *w);
```

- `m` The number of rows in the matrix A.
- `n` The number of columns in the matrix A.
- `ncols` Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.
- `a` Array holding the matrix A.
- `u` Array to hold the matrix U.
- `v` Array to hold the matrix V.
- `w` Array to hold the singular values (a one-dimensional vector).

The return value is zero if the decomposition was successful. A non-zero value, k say, indicates that the iteration for the kth singular value failed. A typical use of `SvdCL` might look like

```
/*   /examples/chapter2/svd.c   */
#include <catam.h>
int MainCL(void)
{
  double a[2][3];
  double u[2][2];
  double v[3][3];
  double w[3];
  int ok;
  a[0][0]=1; a[0][1]=1; a[0][2]=2;
  a[1][0]=0; a[1][1]=2; a[1][2]=2;
  ok=SvdCL(2,3,3,a,u,v,w);
  return 0;
}
```

Eigenvalues and Eigenvectors: the Power method, `EigenMaxCL`

`EigenMaxCL` finds an eigenvalue of largest modulus of a matrix A, together with a corresponding eigenvector.

```
double EigenMaxCL (int n,
                   int ncols,
                   double acc,
                   double *a,
                   double *evec);
```

`n` The order of the matrix.

`ncols` Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.

`acc` The absolute accuracy required.

`Ap` Array holding the matrix A.

`evec` Array holding an initial 'guess' v at the desired eigenvector. This guess does not have to be good, but if it is way out (and you are also very unlucky) you might have $A^n v = 0$ for some n, at which point `EigenMaxCL` will fall over (by setting `ErrorFlagCD=true` and terminating). Assuming this does not happen, when `EigenMaxCL` returns, it will hold a genuine eigenvector.

The return value is an eigenvalue of largest modulus A typical use of `EigenMaxCL` might look like

```
/*   /examples/chapter2/power.c  */
#include <catam.h>
int MainCL(void)
{
  double a[2][2];
  double evect[2];
  double eval;
  a[0][0]=1; a[0][1]=1;
  a[1][0]=1; a[1][1]=0;
  evect[0]=1;
  evect[1]=1;
  eval=EigenMaxCL(2,2,0.00001,a,evect);
  return 0;
}
```

Eigenvalues and Eigenvectors: the Modified Power method, `EigenValCL`

The drawback of `EigenMaxCL` is that it can only give an eigenvector corresponding to an eigenvalue of largest modulus. `EigenValCL` takes a parameter θ and returns an eigenvalue λ and a corresponding eigenvector minimising $|\lambda - \theta|$ for a given matrix A.

```
double EigenValCL (int n,
                   int ncols,
                   double acc,
                   double *a,
                   double *evec,
                   double theta);
```

`n` The order of the matrix A.

`ncols` Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.

`acc` The absolute accuracy required.

`a` Array holding the matrix A.

`evec` Array holding an initial 'guess' v at the desired eigenvector. (This guess does not have to be very good.)

`theta` The parameter θ. If `EigenValCL` encounters a problem (as it will, for example, if θ actually is an eigenvalue) it will set `ErrorFlagCD=true` and terminate.

See `EigenMaxCL` for an illustration of how to use a very similar function. The return value is the desired eigenvalue, λ.

Eigenvalues and Eigenvectors: the Jacobi method for real symmetric matrices, JacobiCL

The Jacobi method is a powerful technique for finding the eigenvalues and eigenvectors of a real symmetric matrix A.

```
void JacobiCL (int n,
               int ncols,
               double *a,
               double *evals);
```

`n` The order of the matrix A.

`ncols` Normally `ncols=n`. Specifically, `ncols` is the size of the second dimension in the declaration of the array `a`.

`a` Array holding the matrix A. This array will be corrupted by the call to `JacobiCL`. The eigenvectors will be returned as the rows of the array `a`.

`evals` Array to hold the eigenvalues.

An example of how to use `JacobiCL` is shown below:

```
/*   /examples/chapter2/jacobi.c   */
#include <catam.h>
int MainCL(void)
{
  double a[2][2];
  double evals[2];
  a[0][0]=1; a[0][1]=-1;
  a[1][0]=1; a[1][1]= 1;
  JacobiCL(2,2,false,a,evals);
  return 0;
}
```

2.4 Special functions

CCATSL provides routines for evaluating three standard mathematical functions, `BesselCL` for Bessel functions of the first kind of integer order, `PhiCL` for the normal distribution function and `InvPhiCL` for its inverse.

Bessel function of first kind, BesselCL

`BesselCL` computes the Bessel function of the first kind, $J_n(x)$ for n a positive integer, which may be defined by

$$J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{1}{2}x\right)^{2k+n}}{k!\,(n+k)!}.$$

```
double BesselCL (int n,
                 double x);
```

`n` Specifies the order n of the required Bessel function.

`x` Specifies the argument to J_n.

The return value is an accurate approximation to $J_n(x)$.

Normal distribution function, PhiCL

`PhiCL` computes the normal distribution function, defined by

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-\tfrac{1}{2}u^2)\, du$$

```
double PhiCL (double x);
```

`x` Specifies the argument to Φ.

The return value is an accurate approximation to $\Phi(x)$.

Inverse cumulative normal density function, InvPhiCL

`InvPhiCL` computes $\Phi^{-1}(x)$, the inverse function to $\Phi(x)$, the normal distribution function.

```
double InvPhiCL (double x);
```

`x` Specifies the argument to Φ^{-1}.

The return value is an accurate approximation to $\Phi^{-1}(x)$.

2.5 FFT and Fast Fourier Sine Transform

CCATSL provides two routines for calculating transforms of discrete sequences, both based on the Fast Fourier Transform. `FftCL` implements the basic Fast Fourier algorithm, while `FftSinCL` computes a Sine transform using `FftCL`.

Fast Fourier Transform, FftCL

`FftCL` computes the Discrete Fourier Transform of a sequence $X_0, X_1, \ldots, X_{N-1}$ of complex numbers, using the Fast Fourier algorithm. The algorithm is an efficient way of calculating the transformed sequence:

$$\widetilde{X}_s = \sum_{r=0}^{N-1} e^{-2\pi isr/N} X_r$$

when $N = 2^m$ for some m. The original sequence can be recovered from the transformed sequence $\widetilde{X}_s$ by

$$X_r = \frac{1}{N} \sum_{s=0}^{N-1} e^{2\pi irs/N} \widetilde{X}_s$$

which is essentially just another application of the FFT algorithm (followed by division by N), but with a change of sign in the exponential.

```
void FftCL (double *x,
            int m,
            double sign);
```

`x` Array holding the sequence $X_0, \ldots, X_{N-1}$. The array `x` points to must be an `double [0..N-1][0..1]` where `x[i][0]` is the real part of X_i and `x[i][1]` is the imaginary part. `FftCL` replaces these values with the transformed sequence, $\widetilde{X}_0, \ldots, \widetilde{X}_{N-1}$. See below for the an example of how to setup a suitable array.

`m` Specifies the number of terms in the sequence; there must be $N = 2^m$ of them.

`sign` The sign of the exponential in the series above (`-1` for the normal transform, `1` for the inverse). Note that when computing the inverse fast Fourier transform by using `FftCL` with `sign=1` you will need to manually divide each resulting X_i by N (see the example below).

Here is an example using `FftCL`:

```
/*   /examples/chapter2/fftex.c   */
#include <catam.h>
int MainCL(void)
{
  int r=3;
  int s;
  double  x[8][2];
  for (s=0; s<8; s++) {
    /* Setup X to be the rth Fourier mode.
       Here the rth Fourier mode, F_s is given by

       F_s = (1/8)exp(2 PI i R s / 8)

       (where i=sqrt(-1)) which we split into
       its real and imaginary parts. */
    x[s][0]=cos(2*M_PI*r*s/8.0)/8.0;
    x[s][1]=sin(2*M_PI*r*s/8.0)/8.0;
    printf("x[%i]=(%f + i %f)\n",s,x[s][0],x[s][1]);
  }
  printf("\n");
  FftCL(x,3,-1.0);  /* transform */
  for (s=0; s<8; s++)
    printf("x[%i]=(%f +i %f)\n",s,x[s][0],x[s][1]);
  FftCL(x,3,1.0);  /* transform back*/
  printf("\n");
  for (s=0; s<8; s++)
    printf("x[%i]=(%f +i %f)\n",s,x[s][0]/8.0,x[s][1]/8.0);
  return 0;
}
```

Fast Fourier Sine Transform, `FftSinCL`

`FftSinCL` computes the sine transform of a sequence $X_0, X_1, \ldots, X_{N-1}, X_N$ of real numbers with $X_0 = X_N = 0$, where $N = 2^m$ for some m:

$$\widetilde{X}_s = \sum_{r=0}^{N-1} \sin(\pi r s/N) X_r.$$

It can also compute the inverse transform. (Note that unlike `FftCL`, when `FftSinCL` performs the inverse transform, no subsequent division by N is necessary.)

```
void FftSinCL (int m,
               double *x,
               double dirn);
```

`m` Specifies the number of terms in the sequence; since $N = 2^m$, there will be $2^m + 1$ terms.

`x` Array holding the sequence $X_0, \ldots, X_N$. `FftSinCL` will replace these values with the transformed sequence.

`dirn` Indicates whether the normal sine transform (`dirn=1`) or the inverse transform (`dirn=-1`) is required.

A typical use of `FftSinCL` might look like:

```
/*   /examples/chapter2/sinfftex.c  */
..
{
  double x[256];
  /* setup the array x */
..
  FftSinCL(8,x,1.0);
}
```

A demonstration of `FftSinCL` can be found in `b08poisn.c`.

2.6 Poisson solver, PoissonCL

CCATSL provides a routine for solving Poisson's equation

$$\nabla^2\psi(x,y) = \zeta(x,y)$$

in a two-dimensional rectangular region, where ψ is specified on the boundary.

```
void PoissonCL (double *psi,
                double *zeta,
                int nx,
                int ny,
                double dlx,
                double dly);
```

`psi` Array to hold the solution ψ. The values on the boundary of the array should be set to the boundary conditions; the rest of the array need not be initialised.

`zeta` Array specifying the function ζ.

`nx` The number of divisions in the lattice in the x-direction.

`ny` The number of divisions in the lattice in the y-direction. (`ny` must be a power of 2.)

`dlx, dly` The grid spacing in the x and y directions respectively.

A typical use of `PoissonCL` might look like

```
/*   /examples/chapter2/poisnex.c   */
#include <catam.h>
int MainCL(void)
{
  int nx=10;
  int ny=8;
  double dlx=0.1;
```

```
  double dly=0.125;
  double psi[nx][ny];
  double zeta[nx][ny];

  /* setup zeta */
  /* .. */

  PoissonCL(psi,zeta,nx,ny,dlx,dly);
  return 0;
}
```

A complete example using `PoissonCL` can be found in `b09poisn.c`.

2.7 Minimisation and root-finding

CCATSL contains three functions for finding the minima and zeros of a real-valued function: `MinCL` finds the minima of a unimodal function in a specified interval, `CubicRootsCL` solves a general cubic equation and `ZeroCL` solves $f(x) = 0$ in a given interval when f is monotone.

Minimisation of a unimodal function, MinCL

`MinCL` finds the location of the minimum of a unimodal function in a given interval, using a method combining golden section search and parabolic interpolation.

```
double MinCL (double (*f)(double x),
              double a,
              double b,
              double tol);
```

`f`	A user-defined function taking a single `double` argument and returning a `double`, giving the value of the function f at the requested point.
`a`	The lower endpoint of the search interval.
`b`	The upper endpoint of the search interval.
`tol`	The acceptable tolerance in the location of the minimum.

A typical use of `MinCL` might look like

```
/*   /examples/chapter2/fminex.c   */
#include <catam.h>
double f(double x)
{
  return 2.1+(x-2.3)*(x-2.3);
}

int MainCL(void)
{
  double ans=MinCL(f,0,6,1e-5);
  printf("minimum occurs at %f\n",ans);
  return 0;
}
```

Cubic equation solver, `CubicRootsCL`

`CubicRootsCL` solves the cubic equation $x^3 + ax^2 + bx + c = 0$. It returns the number of roots, and the roots in non-decreasing order.

```
void CubicRootsCL (double a,
                   double b,
                   double c,
                   int *nroots,
                   double *r1,
                   double *r2,
                   double *r3);
```

`a, b, c`	The coefficients in the cubic equation $x^3 + ax^2 + bx + c = 0$.
`nroots`	Pointer to a variable to hold the number of roots when `CubicRootsCL` returns.
`r1, r2, r3`	Pointers to variables to hold the roots, in non-decreasing order.

Finding the zero of a function, `ZeroCL`

The CCATSL routine `ZeroCL` locates the root of an equation of the form $f(x) = 0$ when f is a monotone function, using a combination of bisection and more powerful methods.

```
double ZeroCL (double (*f)(double x),
               double a,
               double b,
               double tol);
```

`f`	A user-defined function taking a `double` argument and returning a `double`, giving the value of the function f at the requested point.
`a`	The lower endpoint of the search interval.
`b`	The upper endpoint of the search interval.
`tol`	The acceptable tolerance in the location of the root.

Finding the roots of a polynomial, `PolyRootsCL()`

The CCATSL routine `PolyRootsCL` finds all the roots of a polynomial with real coefficients by employing the Laguerre's Method.

```
void PolyRootsCL(float *rcoeff,
                 int deg,
                 complex *roots,
                 int polish);
```

`rcoeff`	vector with the coefficients of the polynomial. These must be real.
`deg`	degree of the polynomial.
`roots`	complex array with the computed roots.
`polish`	set to 1 for polishing the roots, 0 otherwise.

2.8 Spline interpolation

Determining a cubic spline interpolant, SplineCL

`SplineCL` determines the coefficients for a cubic spline interpolating function from a sequence of data-points $(x_i, y_i)_{i=1}^n$. It determines the coefficients $(b_i, c_i, d_i)_{i=1}^n$ in the piecewise cubic interpolating function:

$$y(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad \text{for } x \in [x_i, x_{i+1}).$$

(The coefficients b_n, c_n and d_n are not used in the interpolating function but are used as workspace by `SplineCL`.) The interpolating function passes through the data points and has a continuous first and second derivative everywhere. Once calculated, the interpolating function can be evaluated using `SplineValCL`.

```
void SplineCL (int n,
               double *x,
               double *y,
               double *b,
               double *c,
               double *d);
```

- `n` The number of data points.
- `x` Array holding the sequence $x_1, \ldots, x_n$.
- `y` Array holding the sequence $y_1, \ldots, y_n$.
- `b` Array to hold the coefficients $b_1, \ldots, b_n$.
- `c` Array to hold the coefficients $c_1, \ldots, c_n$.
- `d` Array to hold the coefficients $d_1, \ldots, d_n$.

The example program `b04splin.c` demonstrates the use of `SplineCL`.

Evaluating the interpolant, SplineValCL

Once `SplineCL` has been used to calculate the cubic spline interpolating function for a dataset $(x_i, y_i)_{i=1}^n$, the routine `SplineValCL` can be used to evaluate it at an arbitrary point.

```
double SplineValCL (int n,
                    double x,
                    double *xpts,
                    double *y,
                    double *b,
                    double *c,
                    double *d);
```

- `n` The number of data points.
- `x` The point at which the interpolating function is to be evaluated.
- `xpts` Array holding the sequence $x_1, \ldots, x_n$.
- `y` Array holding the sequence $y_1, \ldots, y_n$.
- `b` Array to hold the coefficients $b_1, \ldots, b_n$.
- `c` Array to hold the coefficients $c_1, \ldots, c_n$.
- `d` Array to hold the coefficients $d_1, \ldots, d_n$.

The example program `b04splin.c` demonstrates the use of `SplineValCL`.

Chapter 3

Plotting graphs

CCATSL contains a large number of routines for plotting graphs of various types. These include routines for two-dimensional data (Section 3.1), such as curves and collections of points and simple histograms, and others for three-dimensional data (Section 3.2), such as contour plots and surface plots. Once you have plotted your graph, CCATSL also provides several ways of printing it out (Section 3.5).

The standard CCATSL plotting functions are normally sufficient for the sort of output that CATAM projects require, but sometimes you will want (or need) to customise some aspect of the graph-plotting process. For example, if you want two or more graphs on screen at once, or you need to draw your graph line by line, or maybe you just want to change the colours, you will have to use additional CCATSL routines: various ways of customising your graph or changing where it appears are described in Section 3.3, while methods for drawing graphs line by line are described in Section 3.4.

3.1 Two-dimensional data

CCATSL has several routines for plotting two-dimensional data. `CurveCL` is the usual choice for plotting $y = g(x)$ when you are able to work out what $g(x)$ will be for arbitrary x. Sometimes you only know the value of $g(x)$ at a sequence of points $(x_i)_{i=1}^n$ when you probably want to use the routine `XYCurveCL` instead. (This routine is also the way to plot arbitrary collections of points.) Finally, CCATSL can draw histograms, using `XYHistogramCL`.

For some of these routines, you may find it helpful to use `XYSortCL` (Section 6.6) to re-order the arrays storing the x and y-values so that the x-values are in increasing order.

CurveCL

`CurveCL` is the simplest CCATSL graphics routine and is suitable for drawing a curve of the form $y = g(x)$ when $g(x)$ is easily computable.

```
void CurveCL (double (*g)(double x),
              double xlow,
              double xhi,
              int npts,
              ColourCT colour,
              AxisModeCT axismode);
```

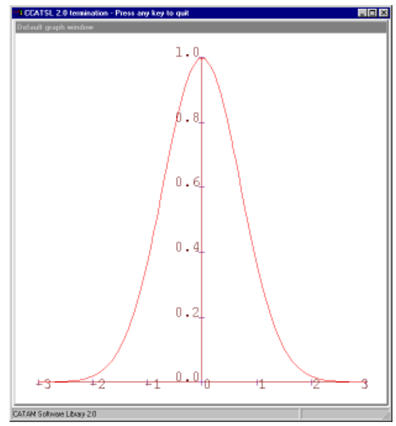
`g`	The function g. See below for an example.
`xlo, xhi`	Specifies the interval `[xlo,xhi]` over which $g(x)$ is to be plotted.
`npts`	The number of points at which g should be sampled.
`colour`	The colour of the graph, .
`axismode`	Determines whether CCATSL should try to work out scales for the x and y axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for more information.

Here is a simple example using `CurveCL`:

```
/*   /example/chapter3/dcurve1.c  */
#include <catam.h>

double f(double x)
{
  return exp(-x*x);
}

int MainCL(void)
{
  CurveCL(f,-3,3,50,RedCC,AUTOAXES);
  return 0;
}
```



XYCurveCL

This routine plots a sequence of points $(x_i, y_i)_{i=1}^{n}$. The points can be plotted individually using a variety of symbols (dots, plus signs, triangles etc.), or they may be joined together with straight lines.

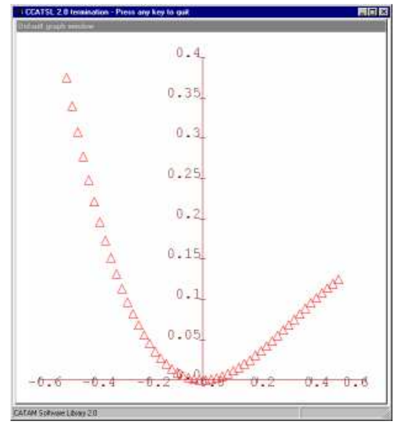
```
void XYCurveCL (double *xp,
                double *yp,
                int npts,
                int ncols,
                DrawDataCT option,
                ColourCT colour,
                AxisModeCT axismode);
```

`xp`	Array holding the sequence $x_1, \ldots, x_n$.
`yp`	Array holding the sequence $y_1, \ldots, y_n$.
`npts`	The number of points in the sequence.
`ncols`	Normally `ncols=1`. (In general, `ncols` should be the size of the second dimension in the declarations of the arrays `xp` and `yp`.)
`option`	Specifies how the points should appear. Popular choices are `JOIN`, to have the points joined up with straight lines, or `PLUS` to have each point marked with a `+`. You can find the full list of plot symbols in Section 7.2.
`colour`	The colour of the graph, such as `RedCC`.

`axismode` Determines whether CCATSL should try to work out scales for the axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for more information.

```
/*   /examples/chapter3/ddata1.c   */
#include <catam.h>

int MainCL(void)
{
  double x[50];
  double y[50];
  double r;
  int i;
  for (i=0; i<50; i++) {
    r=i/49.0-0.5;  /* from -0.5 to +0.5 */
    x[i]=r;
    y[i]=r*r*(1-r);
  }
  XYCurveCL(x,y,50,1,TRIANGLE,RedCC,AUTOAXES);
  return 0;
}
```



XYHistogramCL

This routine produces a two-dimensional histogram from a dataset; the data must be in the form $(x_i, y_i)_{i=1}^n$ where x_i is the location of the ith bar and y_i is its corresponding height. (The centre of the ith bar will be at a position equal to x_i plus some specified offset, which can be arbitrary but must be the same for all i.) The x_i should be equally spaced and monotone—you may want to use the routine `XYSortCL` to sort the arrays first.

```
void XYHistogramCL (double *xp,
                    double *yp,
                    int n,
                    int ncols,
                    double width,
                    double disp,
                    ColourCT colour,
                    AxisModeCT axismode);
```

`xp` Array holding the sequence $x_1, \ldots, x_n$.

`yp` Array holding the sequence $y_1, \ldots, y_n$.

`n` The number of points in the sequence.

`ncols` Normally `ncols=1` (see below). (In general, `ncols` is the size of the second dimension in the declarations of the arrays `xp` and `yp`.)

`width` The width of the bars as a proportion of (the common value of) $x_{i+1} - x_i$. For example, `width=1` will make each bar appear flush with its neighbours while `width=0.9` will leave a small gap between adjacent bars.

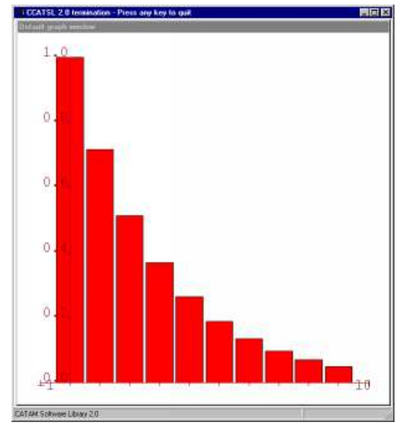
`disp` Specifies the offset of the ith bar relative to x_i, in units of $x_{i+1} - x_i$. For example, `disp=-0.5` places the centre of the ith bar at x_i while `disp=0` will put the left hand side of the ith bar at x_i.

`colour` The colour used to fill the bars, such as `RedCC`.

`axismode` Determines whether CCATSL should work out scales for the axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for more information.

```
/*   /examples/chapter3/histo2d.c  */
#include <catam.h>

int MainCL(void)
{
  double x[10];
  double y[10];
  int i;
  for (i=0; i<10; i++) {
    x[i]=i;
    y[i]=exp(-i/3.0);
  }
  XYHistogramCL(x,y,10,1,0.9,-0.5,RedCC,AUTOAXES);
  return 0;
}
```



3.2 Three-dimensional data

CCATSL provides a large number of routines for plotting three-dimensional data. The simplest is `XYZCurveCL` which plots a sequence of points $(x_i, y_i, z_i)_{i=1}^n$. It can either plot the points individually, or it can join up the points with straight lines.

Surfaces can be plotted using the routine `XYZSurfaceCL`, or as a contour plot using the function `XYZContourCL`. If the function $z(x, y)$ is easy to define in terms of x and y, you can use the simpler routine `ContourCL`, which samples z appropriately. `PolarContourCL` performs a similar function to `XYZContourCL` for functions which are more naturally expressed in polar coordinates: the data is now a collection of points $(r, \theta, z(r, \theta))$ where (r, θ) lie in a (rectangular) lattice. For histograms with two independent variables, CCATSL provides the routine `XYZHistogramCL`.

For some of these routines, you may find it helpful to use `XYZSortCL` (see Section 6.6) to re-order the arrays storing the x, y and z-values so that the x and y-values are in increasing order.

XYZCurveCL

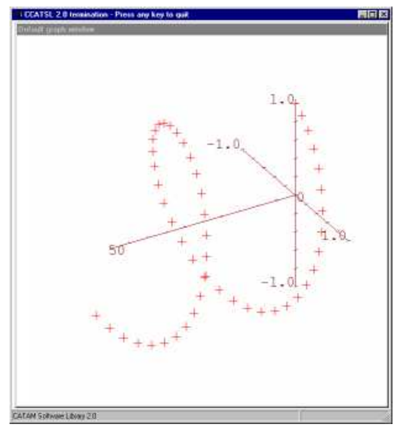
This routine plots a sequence of points $(x_i, y_i, z_i)_{i=1}^n$. The points can be plotted individually using a variety of symbols (dots, plus signs, triangles etc.), or they may be joined together with straight lines.

```
void XYZCurveCL (double *xp,
                 double *yp,
                 double *zp,
                 int npts,
                 int ncols,
                 DrawDataCT option,
                 ColourCT colour,
                 AxisModeCT axismode);
```

`xp`	Array holding the sequence $x_1, \ldots, x_n$.
`yp`	Array holding the sequence $y_1, \ldots, y_n$.
`zp`	Array holding the sequence $z_1, \ldots, z_n$.
`npts`	The number of points in the sequence.
`ncols`	Normally `ncols=1`. (`ncols` is the size of the second dimension in the declaration of the arrays `xp`, `yp` and `zp`.)
`option`	Specifies how the points should appear. Popular choices are `JOIN`, to have the points joined up with straight lines or `PLUS` to have each point marked with a `+`. You can find the full list of plot symbols in Chapter 7.2.
`colour`	The colour of the graph, such as `RedCC` for example.
`axismode`	Determines whether CCATSL should try to work out scales for the axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for further information.

```
/*   /examples/chapter3/ddata3.c   */
#include <catam.h>

int MainCL(void)
{
  double x[50];
  double y[50];
  double z[50];
  int i;
  for (i=0; i<50; i++) {
    x[i]=i;
    y[i]=sin(i/5.0);
    z[i]=cos(i/5.0);
  }
  XYZCurveCL(x,y,z,50,1,PLUS,RedCC,AUTOAXES);
  return 0;
}
```



XYZSurfaceCL

`XYZSurfaceCL` produces surface or wireframe plots from a collection of points of the form $(x, y, z(x, y))$, where (x, y) ranges over a rectangular lattice. If your surface is complicated, it may be better to do a contour plot, using `XYZContourCL`.

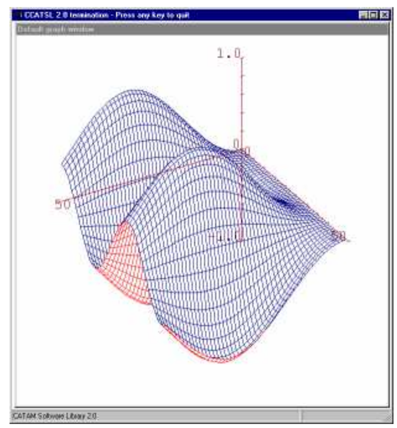
```
void XYZSurfaceCL (double *xp,
                   double *yp,
                   double *zp,
                   int nx,
                   int ny,
                   int ncols,
                   int ncolsZ,
                   DrawObjectCT option,
                   ColourCT upper_col,
                   ColourCT lower_col,
                   AxisModeCT axismode);
```

`xp`	Array holding the sequence of x-values.

`yp`	Array holding the sequence of y-values.
`zp`	Array holding the z-values. (This will normally have been declared with the declaration `double zp[nx][ny]`.)
`nx`	The number of x-values.
`ny`	The number of y-values.
`ncols`	Normally `ncols=1`. (`ncols` is the size of the second dimension in the declaration of the arrays `xp` and `yp`.)
`ncolsZ`	Normally `ncolsZ=ny`. (`ncolsZ` is the size of the second dimension in the declaration of the array `zp`.)
`option`	Can be either `WIREFRAME` for a wireframe plot or `SURFACE` for a hidden line surface plot.
`upper_col`	The colour for the upper side of the surface.
`lower_col`	The colour for the lower side of the surface.
`axismode`	Determines whether CCATSL should try to work out scales for the axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for further information.

```
/*   /examples/chapter3/surface.c   */
#include <catam.h>

int MainCL(void)
{
  double x[50];
  double y[50];
  double z[50][50];
  int i;
  int j;
  for (i=0; i<50; i++) {
    x[i]=i;
    y[i]=i;
    for (j=0; j<50; j++) {
      z[i][j]=sin(i/20.0)*sin(i/20.0)*cos(j/5.0);
    }
  }
  XYZSurfaceCL(x,y,z,50,50,1,50,SURFACE,
                BlueCC,RedCC,AUTOAXES);
  return 0;
}
```



XYZContourCL

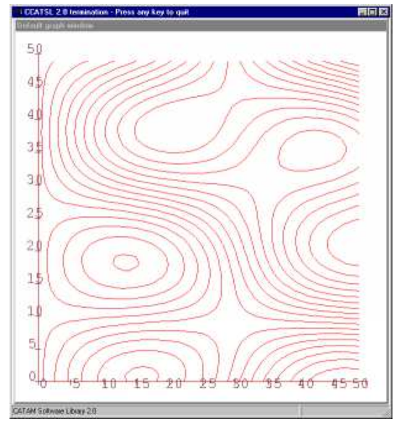
`XYZContourCL` produces contour plots from a collection of points of the form $(x, y, z(x, y))$, where (x, y) ranges over a rectangular lattice.

```
void XYZContourCL (double *xp,
                   double *yp,
                   double *zp,
                   int nx,
                   int ny,
                   int ncols,
                   int ncolsZ,
                   int ncontours,
                   double zlow,
                   double zhi,
                   DrawObjectCT option,
                   ColourCT colour,
                   AxisModeCT axismode);
```

`xp`	Array holding the sequence of x-values.
`yp`	Array holding the sequence of y-values.
`zp`	Array holding the z-values. (This will normally have been declared with the declaration `double zp[nx][ny]`.)
`nx`	The number of x-values.
`ny`	The number of y-values.
`ncols`	Normally `ncols=1`. (`ncols` is the size of the second dimension in the declaration of the arrays `xp` and `yp`.)
`ncolsZ`	Normally `ncolsZ=ny`. (`ncolsZ` is the size of the second dimension in the declaration of the array `zp`.)
`ncontours`	Specifies the number of contours to be drawn.
`zlow`, `zhi`	Specify the range `[zlow,zhi]` of z-values which should produce contours. If you set both to zero, `XYZContourCL` will try to work out a suitable range.
`option`	Either `CONTOURS2D` or `CONTOURS3D`. The first case leads to a flat ‘map’ with contours, the other leads to three-dimensional contour plot.
`colour`	The colour for the contours.
`axismode`	Determines whether CCATSL should try to work out scales for the axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for more information.

```
/*   /examples/chapter3/contour.c   */
#include <catam.h>

int MainCL(void)
{
  double x[50];
  double y[50];
  double z[50][50];
  int i;
  int j;
  for (i=0; i<50; i++) {
    x[i]=i;
    y[i]=i;
    for (j=0; j<50; j++) {
      z[i][j]=sin(i/10.0)*cos(j/6.0)+i*j/800.0;
    }
  }
  XYZContourCL(x,y,z,50,50,1,50,25,0,0,CONTOURS2D,
               RedCC,AUTOAXES);
  return 0;
}
```



ContourCL

This routine provides a simpler way than `XYZContourCL` of producing contour plots of a function $g(x, y)$ when g is easy to evaluate.

```
void ContourCL (double (*g)(double x, double y),
                double xlow,
                double xhi,
                double ylow,
                double yhi,
                double zlow,
                double zhi,
                int ncontours,
                int gtype,
                ColourCT colour,
                AxisModeCT axismode);
```

`g` A user-defined function taking two `double` arguments and returning a `double`. The function should interpret the arguments as x and y respectively and return $g(x, y)$.

`xlow, xhi` The range `[xlow, xhi]` of x-values to plot.

`ylow, yhi` The range `[ylow, yhi]` of y-values to plot.

`zlow, zhi` The range `[zlow, zhi]` of z-values to produce contours.

`ncontours` The number of contours to plot.

`gtype` Specifies the type of contour plot: `gtype=0` produces monochrome contour lines of colour `colour`, `gtype=1` produces contour lines of different colours, `gtype=2` shades the regions between contour lines, and `gtype=3` draws the contour lines in `colour` and shades the region between contour lines.

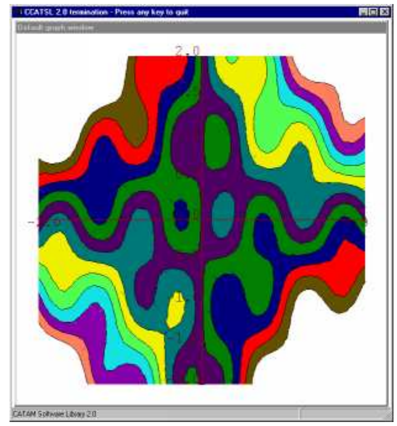
`colour` The colour of the contours.

`axismode` Determines whether CCATSL should try to work out scales for the axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for further information.

```
/*   /examples/chapter3/fncont.c   */
#include <catam.h>

double f(double x, double y)
{
  return sin(2*M_PI*x)*cos(M_PI*y)+2*x*y;
}

int MainCL(void)
{
  ContourCL(f,-2,2,-2,2,-3,5,50,3,RedCC,AUTOAXES);
  return 0;
}
```



PolarContourCL

`PolarContourCL` produces contour plots from a collection of points of the form $(r, \theta, z(r, \theta))$, for $r = r_1, r_2, \ldots, r_n$, $\theta = \theta_1, \theta_2, \ldots, \theta_n$, where (r, θ) are interpreted as polar coordinates.

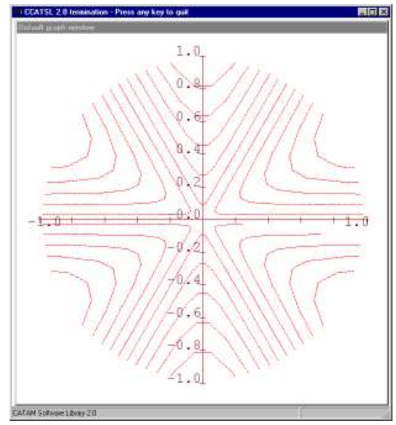
```
void PolarContourCL (double *rp,
                     double *tp,
                     double *zp,
                     int ncolsZ,
                     int nr,
                     int ntheta,
                     double zlow,
                     double zhi,
                     int ncontours,
                     int gtype,
                     ColourCT colour,
                     AxisModeCT axismode);
```

`rp`	Array holding the sequence of r-values.
`tp`	Array holding the sequence of θ-values.
`zp`	Array holding the z-values. (This will normally have been declared with the declaration `double zp[nx][ny]`.)
`ncolsZ`	Normally `ncolsZ=ntheta`. (`ncolsZ` is the size of the second dimension in the declaration of the array `zp`.)
`nr`	The number of r-values.
`ntheta`	The number of θ-values.
`zlow`, `zhi`	The range `[zlow, zhi]` of z-values to produce contours. You can set these both to zero to get CCATSL to work out a sensible range for you.
`ncontours`	Specifies the number of contours to draw.
`gtype`	Specifies the type of contour plot: `gtype=0` produces monochrome contour lines of colour `colour`, `gtype=1` produces contour lines of different colours, `gtype=2` shades the regions between contour lines, and `gtype=3` draws the contour lines in `colour` and shades the region between contour lines.
`colour`	The colour for the contours.

`axismode` Determines whether CCATSL should try to work out scales for the axes automatically. See the example below for the most common usage (`AUTOAXES`), and Section 7.2 for further information.

```
/*   /examples/chapter3/polcont.c   */
#include <catam.h>

int MainCL(void)
{
  double r[50];
  double theta[50];
  double z[50][50];
  int i;
  int j;
  for (i=0; i<50; i++) {
    r[i]=i/50.0;
    for (j=0; j<50; j++) {
      theta[j]=2*M_PI*j/50.0;
      z[i][j]=r[i]*sin(3*theta[j]);
    }
  }
  PolarContourCL(r,theta,z,50,50,50,0,0,15,0,
                 RedCC,AUTOAXES);
  return 0;
}
```



XYZHistogramCL

This routine produces histograms for datasets with two independent variables. The data must be in the form of two sequences (x_i) and (y_i), and an array H_{ij}. The centre of the ijth bar will be located at (x_i, y_i), and its height will be H_{ij}. The sequence (x_i) should be monotone and equally spaced, as should the sequence (y_j).

```
void XYZHistogramCL (double *xp,
                     double *yp,
                     double *zp,
                     int nx,
                     int ny,
                     int nz,
                     int ncols,
                     int ncolsZ,
                     double xwidth,
                     double ywidth,
                     ColourCT colour,
                     AxisModeCT axismode);
```

`xp` Array holding the sequence of x-values.

`yp` Array holding the sequence of y-values.

`zp` Array holding H_{ij}-values. (This will normally have been declared with the declaration `double zp[nx][ny]`.)

`nx` The number of x-values.

`ny` The number of y-values.

`ncols` Normally `ncols=1`. (`ncols` is the size of the second dimension in the declaration of the arrays `xp` and `yp`.)

`ncolsZ` Normally `ncolsZ=ny`. (`ncolsZ` is the size of the second dimension in the declaration of the array `zp`.)

`xwidth` The x-width of the bars as a proportion of (the common value of) $x_{i+1} - x_i$.

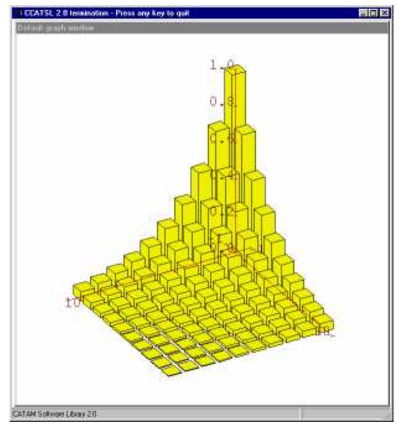
`ywidth` The y-width of the bars as a proportion of (the common value of) $y_{i+1} - y_i$.

`colour` The colour for the bars.

`axismode` Determines whether CCATSL should try to work out scales for the axes automatically. See the example below for the most common usage, and Section 7.2 for more information.

```
/*   /examples/chapter3/histo3d.c   */
#include <catam.h>

int MainCL(void)
{
  double x[10];
  double y[10];
  double z[10][10];
  int i;
  int j;
  for (i=0; i<10; i++) {
    x[i]=i;
    y[i]=i;
    for (j=0; j<10; j++) {
      z[i][j]=exp(-(i+j)/3.0);
    }
  }
  XYZHistogramCL(x,y,z,10,10,1,10,0.8,0.8,
                 YellowCC,AUTOAXES);
  return 0;
}
```



3.3 Customising your graph

This section describes what you can do if the high-level graphics routines of the previous sections don't do quite what you want. Section 3.3.1 explains how to change where your graph appears: drawing graphs in arbitrary CCATSL windows, or in a different part of the default graph window. Section 3.3.2 deals with ways of customising the appearance of the graph itself, and in Section 3.3.3 we describe how to add extra information to your graph such as axis labels and annotations.

3.3.1 Changing where your graph appears

The output of a CCATSL graphics routine appears in the *graph window*, and within this window, inside the *graph port*. If no windows exist when CCATSL tries to draw a graph, a *default graph window* will be created automatically, and the graph port set to the whole window.

If you only want one graph on screen at once, you can forget about graph ports and graph windows and just re-use the default graph window. Should you decide to do this, you may find it helpful to know about the function `PauseCL`

```
PauseCL();        /* Wait for the user to press a key,
                     giving them a chance to printout the graph. */
```

to allow you to make printouts (see Section 3.5 to find out about making printouts), and `WClearCL` which clears the graphics window before drawing your next graph.

If you want several plots to appear on screen at once, you have two choices: either put them in different parts of the default graph window using `GPortCL`, or put each graph in a new window, with `WCurrentCL`. To use the second method, you must first create your own CCATSL window (see Section 4.1) and then make this the current graph window. For example, suppose your program starts

```
#include <catam.h>
int MainCL(void)
{
  WindowCT w1=WindowCL(0,0,0.5,1);
  WindowCT w2=WindowCL(0.5,0,1,1);
  WCurrentCL(w1);
  WShowCL(w1);
  WTitleCL("My left window");

  WCurrentCL(w2);
  WShowCL(w2);
  WTitleCL("My right window");
  return 0;
}
```

to create and open two windows. You can make `wleft` the current graph window by calling `WCurrentCL` after which graphs will appear in `wright`.

WClearCL

`WClearCL` clears the current window:

```
WClearCL();       /* Clear the current window. */
```

GPortCL

`GPortCL` lets you define a rectangular region in the current graphics window. Subsequent graphics commands will produce their output inside this rectangle.

```
void GPortCL (double left,
              double bottom,
              double right,
              double top);
```

`left`, `bottom`	The coodinates of the bottom-left corner of the new graph port, relative to to the bottom left corner of the current graph window, expressed as proportions of the graph window's width and height.
`right`, `top`	The coodinates of the top-right corner of the new graph port, relative to to the bottom left corner of the current graph window, expressed as proportions of the graph window's width and height.

To draw four graphs in the default graph window, you could use

```
GPortCL(0.0, 0.5, 0.5, 1.0);
..                           /* Draw the first graph */
GPortCL(0.5, 0.5, 1.0, 1.0);
..                           /* Draw the second graph */
GPortCL(0.0, 0.0, 0.5, 0.5);
..                           /* Draw the third graph */
GPortCL(0.5, 0.0, 1.0, 0.5);
..                           /* Draw the fourth graph */
```

WCurrentCL

`WCurrentCL` changes the window in which subsequent graphics output appear, and returns the Id previous graphics window.

```
WindowCT WCurrentCL (WindowCT w);
```

`w`   The id of the new graphics window.

The return value is the Id of the previous graphics window.

For example:

```
WCurrentCL(w);     /* Make w (of type WindowCT) the current
                      graph window.  The graph port is reset to
                      the entire window.                        */
```

3.3.2 Changing its appearance

Some aspects of a graph's appearance such as the colour of a curve or the colour of the bars in a histogram can be changed when you call the CCATSL routine. Another set of simple customisations is governed by the `AxisModeCT` (see below). Other possible modifications include: changing the ranges (`XRangeCL`), the colours (`GAxisColoursCL`, `GBrushStyleCL` and `GLineColourCL`), the line style (`GLineStyleCL`), the line 'mode' (`GLineModeCL`), the number of subdivisions of the axes (`XIntervalsCL`), the graph origin (`XOriginCL`) or the viewing direction for 3D plots (`XViewPointCL`).

AxisModeCT

All the CCATSL plotting routines, like `CurveCL` for example, take as their last argument a variable of type `AxisModeCT`. This must be one of `PRESET`, `RESCALE`, `DRAWAXES` and `AUTOAXES`. The most useful is `AUTOAXES`, which asks CCATSL to work out several things, such as sensible scales for the axes, and good places for the pip marks (small lines on the axes) and pip labels (small numbers appearing next to the pip marks). CCATSL then draws the axes, pip marks, pip labels, axes labels (which by default are blank) and finally the graph itself, all in the current graphics window.

If you would like CCATSL to work out scales for the axes as usual, but only to draw the curve, and none of the axes, pip marks, pip labels or axes labels, you use `RESCALE`.

If you are trying to draw a graph on the same set of axes as a previous graph (so you already have the axes, pip marks, pip labels and axes labels drawn and you don't want the scales recalculated), you use `PRESET`, which just draws the curve.

If you want to specify your own ranges for the x, y or z-axes (perhaps you are only interested in a small part of the graph) you can use `XRangeCL`, `YRangeCL` and `ZRangeCL` to change the scales. You now don't want CCATSL to recalculate the ranges, but you do want the axes etc. drawn. In this case, you use `DRAWAXES`.

XRangeCL, YRangeCL and ZRangeCL

These routines allow you to change the range of values plotted on the x, y and z axes. They all have the same syntax so we will just describe `XRangeCL` below.

```
void XRangeCL (double l,
               double h);
```

`l, h` Specifies the new x range as $[l, h]$.

`XOriginCL`, `YOriginCL` and `ZOriginCL`

These routines allow you to change the origin of the subsequent graphs (i.e. the point where the axes cross). They all have similar declarations so we will just describe `XOriginCL` below.

```
void XOriginCL (double x);
```

`x` The new x-coordinate of the orgin.

`XIntervalsCL`, `YIntervalsCL` and `ZIntervalsCL`

These routines allow you to change the number of intervals into which each axis is subdivided. They all have similar declarations so we will just describe `XIntervalsCL` below.

```
void XIntervalsCL (int n);
```

`n` The new number of subdivisions for the axis.

`ViewPointCL`

`ViewPointCL` changes the point from which 3D perspective plots are viewed.

```
void ViewPointCL (double theta,
                  double phi);
```

`theta` The new (Eulerian) angle θ of the viewing direction.

`phi` The new (Eulerian) angle ϕ of the viewing direction.

`GAxisColoursCL`

This routine allows you to change the colours used to draw the pip marks, the axes, and the axis labels

```
void GAxisColoursCL (ColourCT pipclr,
                     ColourCT axisclr,
                     ColourCT labelclr);
```

`pipclr` The colour for the pip marks.

`axisclr` The colour for the axes.

`labelclr` The colour for the axis labels.

GLineColourCL

This routine changes the main colour used in graph-drawing operations, including plot symbols and grid lines (see `XYSymbolCL`, `GGridCL` and `GBoxCL`).

```
void GLineColourCL (ColourCT clr);
```

`clr` The new graphics colour.

GLineStyleCL

This routine changes the style and width of lines produced by graphics routines.

```
void GLineStyleCL (LineStyleCT style,
                   int width);
```

`style` Possible values are `SOLID_LINE`, `DASH_LINE`, `DOT_LINE`, `DASHDOT_LINE`, `DASHDOTDOT_LINE`, and `NULL_LINE`.

`width` The thickness of the pen, in pixels.

GLineModeCL

This routine changes the way new lines appear when they cross existing lines.

```
void GLineModeCL (LineModeCT n);
```

`n` The new line mode, one of `NORMAL_MODE`, `INVERT_MODE`, `EOR_MODE`, `OR_MODE` or `AND_MODE`.

GBrushStyleCL

This routine changes the colour and style of the brush used in fill operations such as `XYPolygonFillCL`.

```
void GBrushCL (ColourCT clr,
               BrushStyleCT k);
```

`clr` The new brush colour.

`k` See see Section 7.2 for a list of available brush styles.

3.3.3 Graph Decorations

Although CCATSL draws axis labels, they are blank by default. They may be changed with a call to `XAxisLabelCL`. (Note that calls to `WCurrentCL`, and the act of opening the default graph window clear the axis labels.)

You can also mark an arbitrary point in a graph in various ways: with text, using `XYLabelCL` (or with `XYZLabelCL` for 3D plots), with one of the standard plotting symbols using `XYSymbolCL` (or `XYZSymbolCL`), or by cross-wires, with `XYCrossWiresCL` (or with `XYZCrossWiresCL`). Textual labels can also be added interactively, using `GLabelCL`

You can also add several other pieces of extra visual information which may make your graph more informative with `GBoxCL`, `GBorderCL` and `GGridCL`.

XAxisLabelCL, YAxisLabelCL and ZAxisLabelCL

These routines let you change the strings used to label the axes (they are blank by default). They all have a similar declaration so we will just describe XAxisLabelCL below.

```
void XAxisLabelCL (char *s);
```

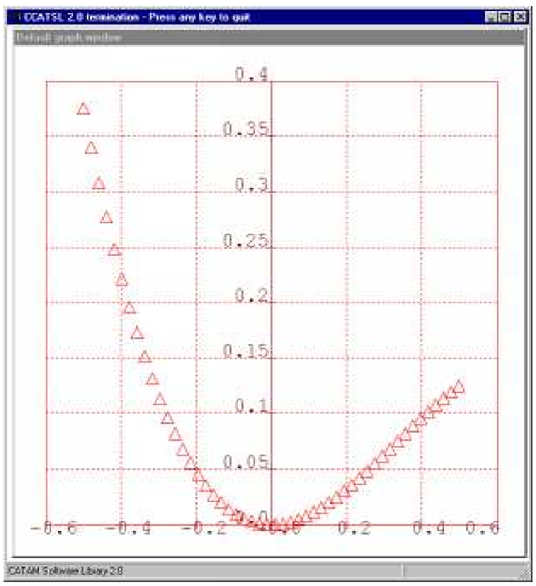
s   The string to use as the axis label.

GBoxCL, GBorderCL and GGridCL

These routines add extra visual information to a graph:

```
GBorderCL();     /* Draw a rectangle around the graph                */
GBoxCL();        /* Draw a box the axes                              */
GGridCL();       /* Draw the box and subdivide it into smaller boxes */
```

Here is an example of the effect of GBoxCL and GGridCL:



The colour may be changed by an immediately preceding call to GLineColourCL.

GLabelCL

GLabelCL gives the user the opportunity to attach a text string to a graph interactively. To label a graph non-interactively, see XYLabelCL for two-dimensional graphs and XYZLabelCL for 3D graphs.

```
GLabelCL();       /* Give the user the chance to label a graph. */
```

XYLabelCL

This routine attaches a textual label to a point on a two-dimensional graph. XYLabelCL performs the analogous function for three-dimensional graphs.

```
void XYLabelCL (double x,
                double y,
                char *txt);
```

`x, y`	The coordinates of the point to be labelled.
`txt`	The text of the label.

XYZLabelCL

This routine attaches a textual label to a point on a three-dimensional graph. `XYLabelCL` performs the analogous function for two-dimensional graphs.

```
void XYZLabelXL (double x,
                 double y,
                 double z,
                 char *txt);
```

`x, y, z`	The coordinates of the point to be labelled.
`txt`	The text of the label.

XYSymbolCL

This routine marks a point on a two-dimensional graph with one of the standard plotting symbols (see Section 7.2). `XYZSymbolCL` performs the analogous function for three-dimensional graphs.

```
void XYSymbolCL (double x,
                 double y,
                 int sym);
```

`x, y`	The coordinates of the mark.
`sym`	The mark, for example `sym=XCROSS` makes the mark an `XCROSS`, and similarly for the other plotting symbols.

XYZSymbolCL

This routine marks a point on a three-dimensional graph with one of the standard plotting symbols (see Section 7.2). `XYSymbolCL` performs the analogous function for two-dimensional graphs.

```
void XYSymbolCL (double x,
                 double y,
                 double z,
                 int sym);
```

`x, y, z`	The coordinates of the mark.
`sym`	The mark, for example `sym=XCROSS` makes the mark an `XCROSS`, and similarly for the other plotting symbols.

XYCrossWiresCL

This routine draws crosswires through a point on a two-dimensional graph. `XYZCrossWiresCL` performs the analogous function for three-dimensional graphs.

```
void XYCrossWiresCL (double x,
                     double y);
```

`x, y` The coordinates of the point.

XYZCrossWiresCL

This routine draws crosswires through a point on a 3D graph. `XYCrossWiresCL` performs the analogous function for two-dimensional graphs.

```
void XYZCrossWiresCL (double x,
                      double y,
                      double z);
```

`x, y, z` The coordinates of the point.

3.4 Drawing graphs line by line

While the high level CCATSL graphics routines make it easy to draw the graph of a function, or to display a dataset in a standard form, it is sometimes better to take control of the graph drawing process yourself and draw each line or point individually. A sequence of commands to draw a triangle might look like:

```
Set2DPlotCL();           /* Select a 2D plot                 */
XRangeCL(-1.0,1.0);      /* Establish the new x-scale        */
YRangeCL(-1.0,1.0);      /* Establish the new y-scale        */
GAxesCL();               /* Draw the axes                    */
XYMoveCL(-1.0,-1.0);     /* Move to the point (-1, -1)       */
XYDrawCL( 0.0, 1.0);     /* Draw a line to the point ( 0, 1) */
XYDrawCL( 1.0,-1.0);     /* Draw a line to the point ( 1,-1) */
XYDrawCL(-1.0,-1.0);     /* Draw a line to the point (-1,-1) */
```

To use this approach, before trying to draw anything, you must establish scales for your axes. This can be done with `XRangeCL` etc. (as in the example), but it also happens automatically when you call one of the high level CCATSL graphics routines with an `AxisModeCT` of `AUTOAXES` or `RESCALE` (see Section 3.3.1). Once the scales have been established, if you have not just called a high-level graphics routine, you will probably first want to draw the axes and axis-labels with `GAxesCL`. After that you can add points using `XYSymbolCL`, lines with `XYMoveCL` and `XYDrawCL`, polygons with `XYPolygonCL` and filled polygons with `XYPolygonFillCL`. The functions `XYZSymbolCL`, `XYZMoveCL`, `XYZPolygonCL` and `XYZPolygonFillCL` provide similar functionality for three-dimensional graphs.

(The routines `XYSymbolCL` and `XYZSymbolCL` are described in Section 3.3.3, while descriptions of `XYPolygonCL`, `XYPolygonFillCL`, `XYZPolygonCL` and `XYZPolygonFillCL` may be found in Section 6.1.)

GAxesCL

This command asks CCATSL to draw and label the axes. You do not normally have to call these routines yourself since CCATSL will draw (or re-draw) and label the axes whenever you use one of the high level graphics routines. You should call either `Set2DPlotCL` or `Set3DPlotCL` prior to calling `GAxesCL`.

```
GAxesCL();   /* draw the axes */
```

`Set2DPlotCL` and `Set3DPlotCL`

These functions tell CCATSL the number of dimension in the next plot.

```
Set2DPlotCL();  /* prepare for a 2D plot */
/* .. */
Set3DPlotCL();  /* prepare for a 3D plot */
```

`XYMoveCL`, `XYDrawCL`, `XYZMoveCL` and `XYZDrawCL`

The routines either draw a line from the current graphics cursor to a specified point or move the graphics cursor to a new point. For two-dimensional graphs, the first two routines should be used. For three-dimensional graphs the final routines must be used. The syntax of these commands is straightforward:

```
XYMoveCL(1.0, 1.0)     /* move the graphics cursor to (1.0, 1.0) */
XYDrawCL(1.0,-1.0)     /* draw a line to (1.0, -1.0)             */
```

for a two-dimensional graph, and

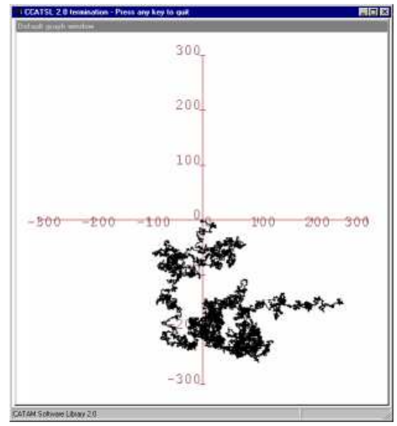
```
XYZMoveCL(1.0, 1.0, 1.0)     /* move the graphics cursor to (1.0, 1.0, 1.0) */
XYZDrawCL(1.0,-1.0, 1.0)     /* draw to (1.0, -1.0, 1.0)                    */
```

for a three-dimensional graph. For example:

```
/*   /examples/chapter3/drawabs.c   */
#include <catam.h>

int MainCL(void)
{
  double x,y;
  int change;
  double dx,dy;

  Set2DPlotCL();
  XRangeCL(-300,300);
  YRangeCL(-300,300);
  GAxesCL();
  x=0.0;
  y=0.0;
  XYMoveCL(0.0,0.0);
  while (x*x+y*y<90000) {
    change=RandomIntCL(4);
    dx=0.0;
    dy=0.0;
    switch (change) {
    case 1: dx= 1; break;
    case 2: dx=-1; break;
    case 3: dy= 1; break;
    case 4: dy=-1; break;
    }
    x=x+dx;
    y=y+dy;
    XYDrawCL(x,y);
  }
  return 0;
}
```



3.5 Printing out graphs

If you want to include your graphs in a word-processed report, the simplest way to proceed is by using Windows's builtin window-capture feature by pressing Alt+PrintScreen (this was how the screenshots for this book were produced). This copies the contents of the active window into the clipboard, and it can then be 'pasted' into your document in the normal way.

A better approach which is more complicated, but more versatile, is to save each graph to a disk file called a metafile. This can be printed out later, or incorporated in a word-processed report. To create and save a metafile containing a graph, use the routines `GRecordCL` and `GSaveCL`. Alternatively, you can do everything from the system menu: select `record graphics` to start recording graphics commands, wait until your program has drawn the graph, then select `Record current window graphics` again, to stop recording. Now select `Save recording as Metafile` to save the recorded graphics. You may find it helpful to call the CCATSL function `PauseCL` at appropriate points in your program.

Once you have all your graphs saved as metafiles, they can be viewed and printed by selecting `display metafile` or `print/display metafile` from the system menu, or by calling `DisplayMetaFilesCL` (below) in your program.

The example program `b10print.c` provides further demonstration.

GRecordCL

This routine 'opens' a metafile, causing subsequent graphics commands to the current graphics window up to the next call to `GSaveCL`, to be recorded in the metafile. See `GSaveCL` for to see how to save the resulting metafile.

```
GRecordCL();                /* start recording graphics commands */
..                          /* commands to produce the graph     */
GSaveCL("mygraph.wmf");     /* stop recording and save the graph */
```

GSaveCL

This routine saves the the sequence of graphics commands since the last `GRecordCL` in a disk file in Enhanced Windows Metafile Format. The file may be subsequently displayed and/or printed using the routine `DisplayMetaFilesCL`.

```
void GSaveCL (char *fnam);
```

`fnam` The filename to save the metafile under. If `fnam=""`, the user will be prompted for a filename.

DisplayMetaFilesCL

`DisplayMetaFilesCL` prints out a collection of metafiles, positioning them in a grid on the page. The user will prompted for the names of the files, (the first may be specified in the call to `DisplayMetaFilesCL`) and for other information.

```
void DisplayMetaFilesCL (int ncols,
                         int nrows,
                         char *filename);
```

`ncols` The number of columns in the grid.

`nrows` The number of rows in the grid.

`filename` The name of the first metafile. This may be `""` in which case the user will be prompted for a filename.

Chapter 4

Using CCATSL Windows

CCATSL provides a simple interface for using windows, and for customising aspects of their appearance such as background colour, text colour and text font. The problem of displaying graphics in a CCATSL window is described in detail in the previous chapter. Here we concern ourselves with textual output.

4.1 Basic CCATSL window routines

CCATSL windows are created with `WindowCL`, or `WindowExCL`, but they must also be opened (made visible) with `WShowCL` before they can be used. Some other useful functions are `WClearCL`, which clears the window, `WHideCL`, which makes the window disappear (it can be restored with another call to `WShowCL` but the contents will need to be redrawn), and `WCurrentCL` (see Section 3.3.1) which changes the window in which CCATSL graphics appear.

WindowCL

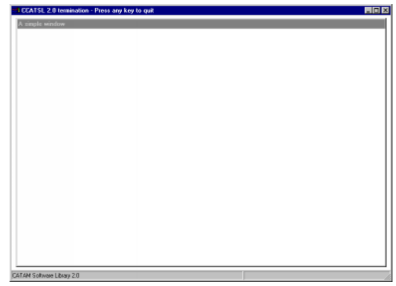
This routine creates a new CCATSL window. The window will not be visible until after a call to `WShowCL`. A more general window creation routine is `WindowExCL`.

```
WindowCT WindowCL (double left,
                   double bottom,
                   double right,
                   double top);
```

`left,` `bottom,` `right, top`	The initial location of the of bottom-left and top-right corners of the window relative to the bottom-left corner of the main CCATSL window, and expressed as proportions of the width and height of the main CCATSL window.

The return value is the id of the new window. Many of the example programs serve as illustrations of `WindowCL` but a minimal example might look like:

```
/*   /examples/chapter4/newwin.c  */
#include <catam.h>
int MainCL(void)
{
  WindowCT w;
  w=WindowCL(0.1,0.1,0.9,0.9);
  WShowCL(w);
  WTitleCL("A simple window");
  return 0;
}
```



WindowExCL

`WindowExCL` is a more general procedure for creating windows than `WindowCL`.

```
WindowCT WindowExCL (double left,
                     double bottom,
                     double right,
                     double top,
                     ColourCT tcol,
                     ColourCT bgcol,
                     ColourCT gcol,
                     WindowTypeCT wtype,
                     int pw,
                     int ph);
```

`left, bottom, right, top`	The initial location of the of bottom-left and top-right corners of the window relative to the bottom-left corner of the main CCATSL window, and expressed as proportions of the width and height of the main CCATSL window.
`tcol, bgcol, gcol`	`tcol` and `bgcol` specify the text colour and the background colour for the window. `gcol` sets the initial graphics line colour
`wtype`	The window type. See Section 7.2 for details.
`pw,ph`	The width and height of the 'bitmap', which may be larger than the actual window. This is only relevant if `wtype=SCROLLINGWIN`.

The return value is the id of the new window.

WShowCL

`WShowCL` makes a window previously created with `WindowCL`, or `WindowExCL`, or closed with `WHideCL`, visible again.

```
void WShowCL (WindowCT w);
```

`w`	The identifier of the window to be opened.

WHideCL

This routine makes a given window disappear. The window can be made visible again using `WShowCL` but the contents will have to be redrawn.

```
void WHideCL (WindowCT w);
```

`w`	The identifier of the window to be hidden.

4.2 Writing in a window

To write text in a CCATSL window, CCATSL provides the function `PrintfCL`.

PrintfCL

`PrintfCL` is the analogue of the standard C library routine `printf` but uses the current CCATSL window. The routine `WLinesCL` can be used to find out how many lines of text a window can hold.

```
void PrintfCL (int col,
               int row,
               char *format, ...  );
```

`col, row`	The location of the start of the string, relative to the top-left corner of the window: `1,1` is the top-left corner.
`format, ...`	See `printf` for an explanation of the remaining arguments.

An example of using `PrintfCL` is shown below. Note that because the exact screen location of the string is specified in the first two arguments, there is no need to append a newline character `\n` to the format string.

```
int i;
double r;

/* ... */

PrintfCL(1,1,"Iteration=%i Radius=%f",i,r);
```

WLinesCL

`WLinesCL` returns the maximum number of text lines viewable in the current window. This is useful in the context of the routine `PrintfCL`.

```
n=WLinesCL();
```

4.3 Changing the appearance of text: colours and fonts

When you create a CCATSL window, you can specify an initial text and background colour; these can be changed at any time with the routine `TextColoursCL`. CCATSL also allows you to change the text font, with `FontCL`.

`TextColoursCL`

This routine changes the current text and background colours for a given window.

```
void TextColoursCL (ColourCT tcol,
                    ColourCT bcol);
```

`tcol` The new text colour.

`bcol` The new background color.

`FontCL`

This routine changes the current font used in the current window.

```
void FontCL (char *fontname,
             int size,
             FontStyleCT style);
```

`fontname` The name of the font, such as 'Arial', 'Times New Roman' or 'Courier New'.

`size` The point size required.

`style` The desired style, see Section 7.2 for details.

4.4 The message, status and error windows

CCATSL uses a standard mechanism for reporting status, error and informational messages. You can use this yourself with `StatusCL`, `ErrorCL` and `MessageCL`.

`MessageCL`, `StatusCL` and `ErrorCL`

These routines allow you to use CCATSL's standard mechanism for reporting status, error, and informational messages.

```
MessageCL("Please select an option");   /* Write to the message area */
StatusCL("Running...");                 /* Write to the status area  */
ErrorCL("Failed to converge");          /* Write error message and exit */
```

4.5 Printing it out

All the techniques for printing out graphs (Section 3.5) can be used to print out CCATSL windows containing text (as far as CCATSL is concerned, text is just a special type of graphic). If you use the standard C library routines `printf` and `scanf` (which write to a special window called the Stdio window) you can use `PrintColourCL`, `PrintCL` and the window-capture method (described in Section 3.5), or you can use `fopen` to write the data to a disk file (Section 1.9.4), and then load up the file and print it out from a text editor (including Emacs). Simpler ways which will work provided you haven't written more than 400 lines are `FileStdioCL` and `PrintStdioCL`.

`FileStdioCL` and `PrintStdioCL`

These routines provide an easy way of saving and printing out the last 400 lines of text from the Stdio window.

```
FileStdioCL("output.txt");   /* Save the Stdio window to a text file.
                                If the argument is "" the user will
                                be prompted for a filename.            */

PrintStdioCL();              /* Printout the contents of the Stdio window */
```

The second of these can be accessed most easily via the system menu (obtained by clicking in the small box at the top-left of the main CCATSL window).

Chapter 5

Getting input from the user

5.1 Entering variables

This section describes more user-friendly alternatives to the standard C library routine `scanf`. For simple variables, CCATSL provides the routines `ReadIntCL`, `ReadDoubleCL`, and `ReadStringCL` which open a dialog box, prompt the user and return the input.

`ReadIntCL`, `ReadDoubleCL` and `ReadStringCL`

These routines open a dialog box, prompt the user, and return the value entered, by returning a variable of an appropriate type: (`int`s for `ReadIntCL`, `double`s for `ReadDoubleCL` and `char*`s for `ReadStringCL`) (see Section 1.4.3 for information about strings and `char*` variables.) The routines have almost identical syntax so we will just describe `ReadIntCL` below:

```
int ReadIntCL (char *p,
               int default);
```

`p` The prompt.

`default` The value to use as default.

The return value is the value entered by the user For example:

```
int n;
double d;
char *name;    /* will hold a string   */

n=ReadIntCL("Enter n ",10);                    /* Read in an int   */
d=ReadDoubleCL("Enter the stepsize ",0.0001);  /* Read in a double */
name=ReadStringCL("Enter your name ","");      /* Read in a string */
```

5.2 Menus

Pull-down menus are a familiar part of most graphical user interfaces, and CCATSL make it very easy to include menus in your own programs. The simplest way is to use `MenuCL`, which creates a menu and then waits for the user to select one of the items. Sometimes you will want several menus side-by-side, each containing logically related functions. In this case you must define each menu separately with `MenuOpenCL` giving each menu a unique *menu ID*, and then call `MenuSelectCL` to obtain the user's selection. Once opened, a menu can be removed with `MenuCloseCL`.

It is sometimes useful to restrict the items that a user can select from your menus. CCATSL allows you to deactivate individual items in a menu (deactivated items will appear 'grayed'), or even an entire menu, using `MenuRestrictCL` and `MenuRestCL`. Deactivated items can be re-activated with `MenuWakeCL`.

Many of the example programs provide illustrations of the menu routines.

MenuCL

This routine creates a single pull-down menu, and then waits for the user to select one of the items.

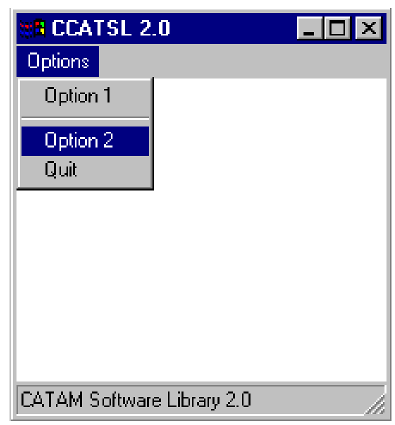
```
int MenuCL (char *caption,
            char *itemList);
```

`caption` The string which appears at the top of the menu. Until the user clicks on it, only `caption` is visible, below which the entire menu will appear.

`itemList` The list of menu items. This should be a `$`-separated list of strings. Each string represents a menu item, except for the special string `(-)` which is interpreted as a separator (a horizontal line dividing menu items). See the example below.

The return value is the index of the selected item (starting from 1).

```
/*   /examples/chapter5/ctmenu.c  */
#include <catam.h>
int MainCL(void)
{
  int opt;
  do {
    opt=MenuCL("Options","Option 1$(-)$Option 2$Quit");
  } while (opt!=4);
  return 0;
}
```



MenuOpenCL

This routine defines a menu, opens it, and associates it with a menu ID. `MenuSelectCL` can be used to give the user the opportunity to select an item from such a menu.

```
void MenuOpenCL (int id,
                 char *caption,
                 char *itemList);
```

`id` The menu ID to associate with the menu. If a menu already exists with the same ID, the previous menu will be destroyed. Valid IDs are in the range 4–20.

`caption` The string to appear at the top of the menu. Only the `caption` is visible until the user clicks on it, when the entire menu will appear beneath.

`itemList` The list of items for the menu. This should be a `$` separated list of string. Each string represents a menu item, except for the special string `(-)` which is interpreted as a separator (a horizontal line dividing menu items).

MenuSelectCL

This routine waits for the user to select one of the items from one of the currently open menus (see `MenuOpenCL`) and returns the corresponding menu ID and item number. Only active menu items can be selected (see `MenuWakeCL`, `MenuRestCL` and `MenuRestrictCL` to see how to restrict the set of possible selections).

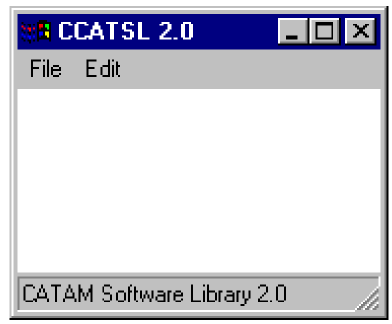
```
void MenuSelectCL (int *id,
                   int *item);
```

`id` Pointer to a variable to hold the menu ID of the selected item when `MenuSelectCL` returns.

`item` Pointer to a variable to hold the index (1, 2, etc.) of the selected item when `MenuSelectCL` returns.

```
/*   /examples/chapter5/select_f.c   */
#include <catam.h>

int MainCL(void)
{
  int nmenu,nitem;
  MenuOpenCL(4,"File","Quit");
  MenuOpenCL(5,"Edit","Paste");
  MenuSelectCL(&nmenu,&nitem);
}
```



MenuCloseCL

This routine destroys an open menu (see `MenuOpenCL`). A typical use of `MenuCloseCL` is to replace one set of menus with another.

```
void MenuCLoseCL (int id);
```

`id` The menu ID of the menu to be closed and deleted.

MenuRestCL

This routine deactivates all the items of a menu, causing `MenuSelectCL` to ignore attempts to select any of its items. The items will appear 'grayed' to the user.

```
void MenuRestCL (int id);
```

`id` The ID of the menu to rest.

MenuWakeCL

This routine makes all items of a menu active again (see also `MenuSelectCL`, `MenuRestCL` and `MenuRestrictCL`.

```
void MenuWakeCL (int id);
```

`id`	The ID of the menu.

`MenuRestrictCL`

This routine activates or deactivates a selection from a menu.

```
void MenuRestrictCL (int id,
                     int item,
                     int control);
```

`id`	The ID of the menu.
`item`	The index (1, 2, etc.) of the item to activate or deactivate.
`control`	`control=0` deactivates the item, `control=1` activates it.

5.3 Entering and Editing arrays

CCATSL provides two useful routines allowing the user to enter and edit arrays of `double`s and `int`s: `EditDoubleArrayCL` and `EditIntArrayCL`.

`EditDoubleArrayCL`

This routines displays a two-dimensional `double` array in a CCATSL window and gives the user the opportunity to edit the entries.

```
void EditDoubleArrayCL (int x,
                        int y,
                        int field,
                        int ndec,
                        two_d_double_array ap,
                        int nrows,
                        int ncols,
                        int nTotalCols);
```

`x, y`	The text-coordinates of the position for the top-left corner of the array.
`field`	The field-width to use when printing each array entry.
`ndec`	The number of decimal places to display.
`ap`	The array of doubles, defined as type `typedef double two_d_double_array[][]`.
`nrows, ncols`	The number of rows and columns to print. Usually `nrows` is the size of the first dimension in the declaration of the array `ap`, and `ncols` the second.
`nTotalCols`	The size of the second dimension in the declaration of the array `ap`.

For example:

```
..
{
  double r[5][10];
  EditDoubleArrayCL(
                  1, 1,       /* print in the top-left corner */
```

```
                      4, 1,       /* field width and decimal places */
                      r,
                      5, 10, 10); /* array dimensions */
}
```

EditIntArrayCL

This routine displays a two-dimensional `int` array in a CCATSL window and gives the user the opportunity to edit the entries.

```
void EditIntArrayCL (int x,
                     int y,
                     int field,
                     two_d_int_array ap,
                     int nrows,
                     int ncols,
                     int nTotalCols);
```

`x, y`	The text-coordinates of the position for the top-left corner of the array.
`field`	The field-width to use when printing each array entry.
`ap`	The array of `int`s, defined as type `typedef int two_d_int_array[][]`.
`nrows, ncols`	The number of rows and columns to print. Usually `nrows` is the size of the first dimension in the declaration of the array `ap`, and `ncols` the second.
`nTotalCols`	The size of the second dimension in the declaration of the array `ap`.

Example: #include ¡catam.h¿ int MainCL(void) int x[2][3] = 0, 1, 2, 10, 11, 12; EditIntArrayCL(1, 1, 5, x, 2, 3, 3);

If the array has only one dimension pass its address: #include ¡catam.h¿ int MainCL(void) int y[12] = 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23; EditIntArrayCL(1, 1, 5, &y, 3, 4, 4); printf("

5.4 The Escape key

Another type of input which is occasionally useful to detect is the hitting of the Escape. This is generally interpreted as some form of 'quitting'. A good place to check for Escape is inside a long loop, such as an iteration which may take a long time to converge. The user may decide that the iteration is not going to give a useful result and would like to restart it with different parameters. Good style would be to use `StatusCL` to tell the user that something is running and `MessageCL` to tell the user how to stop it. You can then test to see if Escape has been hit at some point in the loop:

```
StatusCL("Running...");
MessageCL("Hit Escape to quit")
do {
..
} while (fabs(err)>eps || EscapeCL());
```

Another useful routine is `PauseCL`, which simply waits until a key is pressed.

```
PauseCL();   /* Wait until a key is pressed */
```

EscapeCL

This function returns true if the user has hit the Escape key since the last time `EscapeCL` was called.

```
StatusCL("Running...");
MessageCL("Hit <Esc> to quit")
do {
..
} while (fabs(err)>eps || EscapeCL());
```

PauseCL

This routine displays a message to the user and waits for a key to be pressed.


```
PauseCL();   /* Wait until a key is pressed */
```

Chapter 6

Miscellaneous CCATSL routines

6.1 Graphics

`XYPolygonCL` and `XYPolygonFillCL`

These routines respectively draw and fill a polygonal region, defined by a sequence of points (x_i, y_i). The colour used by `XYPolygonFillCL` can be changed with `GBrushStyleCL`. Since the syntax of the two routines is the same, we will just describe `XYPolygonCL` below. Routines for drawing and shading polygons defined by a sequence of coplanar points in three-dimensions are `XYZPolygonCL` and `XYZPolygonFillCL`.

```
void XYPolygonCL (double *xp,
                  double *yp,
                  int n,
                  int ncols);
```

`xp, yp`	Arrays holding the sequences (x_i) and (y_i).
`n`	The number of points in the sequences.
`ncols`	The size of the second dimension in the declaration of the arrays `xp` and `yp`.

`XYZPolygonCL` and `XYZPolygonFillCL`

These routines draw and fill a polygon, defined by a sequence of coplanar points (x_i, y_i, z_i) in three-dimensions. The colour used by `XYZPolygonFillCL` can be changed with `GBrushCL`. Since the syntax of the two routines is the same, we will just describe `XYZPolygonCL` below.

```
void XYZPolygonCL (double *xp,
                   double *yp,
                   double *zp,
                   int n,
                   int ncols);
```

`xp, yp, zp`	Arrays holding the sequences (x_i), (y_i) and (z_i).
`n`	The number of points in the sequences.
`ncols`	The size of the second dimension in the declaration of the arrays `xp`, `yp` and `zp`.

IsotropicCL

This routine sets equals scales in all three coordinates, returning the adjusted ranges in a `XYZRangesCT` structure.

```
XYZRangesCT t;
t=IsotropicCL();
```

RGBColourCL

This routine constructs a `ColourCT` from a triple of `int`s representing the strengths of the red, green and blue components, and returns a `ColourCT`. The routine will, if required, store the new colour in the `CTColourSet` array, allowing it to be used in CCATSL routines such as `ContourCL`.

```
ColourCT RGBColourCL (int index,
                      int r,
                      int g,
                      int b);
```

`index` The index in the `CTcolorset` array (0–20) to store the new colour. If `index<0` the colour will not be stored.

`r, g, b` The strengths of the red, green and blue components in the new colour. `r`, `g` and `b` should be in the range 0–255.

GDefaultsCL

This resets all the graphics parameters to the default.

```
GDefaultsCL();    /* Reset graphics parameters to the defaults */
```

AspectRatioCL

This function returns the screen's aspect ratio (the number of vertical lines divided by the number of horizontal lines).

```
double AspectRatioCL (int n);
```

`n` Identifies the area to be inspected. If `n=0` the return value refers to the client area of the graphics window, if `n=1` it refers to the entire graphics window (including the title bar etc.), and if `n=2` it refers to the entire screen.

XYMouseCL

Waits for a mouse button to be pressed and returns the information related to the mouse click.

```
int XYMouseCL (double *x,
               double *y);
```

`x,y` Pointers to variables used to receive the location of the mouse click. The mouse must be clicked in the current graphics window. This function does not return sensible values if the current graphics window has a 3D graph!

The return value is 1 if the left mouse button was pressed and 2 if the right mouse button was pressed. If the Escape key is pressed the function returns 0 and `x` and `y` are unchanged. In this case `EscapeCL` should be called to clear the Escape flag before any further calls to `XYMouseCL`.

6.2 Windows, Menus and dialog boxes

WSelectCL

`WSelectCL` changes the graph window without bringing it to the front (helpful if you are drawing graphs in two windows).

```
WSelectCL(w);    /* w has type WindowCT */
```

WTitleCL

`CTwtitle` changes the title of the current graphics window.

```
char* WTitleCL (char *title);
```

`title` The new window title.

The return value is the previous window title.

CursorCL

`CursorCL` changes the cursor (mouse pointer).

```
void CursorCL (CursorStyleCT csr);
```

`csr` The new cursor: `ARROW`, `IBEAM`, `CROSS`, `RESIZE`, `WAIT`, `HIDDEN` and `VISIBLE`.

6.3 Time

TickCL and TockCL

`TickCL` routine resets CCATSL's centi-second timer. `TockCL` returns the number of centi-seconds since the previous call to `TickCL`.

```
TickCL();
....                    /* time this code */
time_taken=TockCL();    /* return type is int */
```

WaitCL

This routine pauses execution of the program for a specified number of seconds. See also `PauseCL`.

```
void WaitCL (double t);
```

`t` The number of seconds to pause for.

6.4 Disk files

FileFindCL and FileNewCL

These routines perform various file-related operations. `FileFindCL` uses a dialog box to prompt the user for the filename of an existing file, `FileNewCL` prompts the user for the name of a new file. The routines have the same syntax so we will only describe `FileFindCL`.

```
boolean FileFindCL (char *pname,
                    char *fname);
```

`pname` A string to receive the fully qualified pathname.

`fname` A string to receive the filename.

The return value is `false` on failure.

FileDeleteCL

This routine deletes a file.

```
success=FileDeleteCL(pname);      /* delete the file with pathname pname
                                     (relative to the current directory) */
```

The return value is `false` on failure.

6.5 Random Numbers

CCATSL provides three functions for controlling the generation of pseudo-random numbers. `RandomCL` returns a random number distributed uniformly on $[0, 1)$, `RandomIntCL` returns a random integer uniformly distributed on the set $\{1, 2, \ldots, n\}$ for specified $n \geq 2$, and `SetRandomCL` allows you to change the current state of the random number generator, useful if you want to ensure your program uses the same sequence of random number each time it is run.

RandomCL

This routine returns a random number uniformly distributed on $[0, 1)$.

```
double u;
...
u=RandomCL();    /* get a U[0,1) rv */
```

RandomIntCL

This routine returns a random integer uniformly distributed on the set $1, 2, \ldots, n$ for specified $n \geq 2$.

```
int RandomIntCL (int n);
```

`n` The size of the set

SetRandomCL

This routine allows you to change the current state of the random number generator. If it is not called, the program will generate the same sequence of random numbers each time it is run

```
void SetRandomCL (int seed);
```

`seed` If `seed>0`, this will set the seed for subsequent random number generations. If `seed<0`, the random number generator will be seeded using a number derived from the current time. Used this way, we can ensure that successive runs of a program use independent sequences of random numbers.

6.6 Miscellaneous

XSortCL

`XSortCL` sorts the entries of a one-dimensional array of `double`s into increasing order.

```
void XSortCL (double *xp,
              int nx);
```

`xp` Array containing the entries to sort.

`nx` The number of elements in the array pointed to by `xp`.

XYSortCL

`XYSortCL` sorts the entries of a one-dimensional array of `double`s into increasing order and applies the same permutation to a second array.

```
void XYSortCL (double *xp,
               double *yp,
               int nx);
```

`xp` Array containing the entries to sort.

`yp` Second array to be re-ordered.

`nx` The number of elements in the arrays `xp` and `yp`.

XYZSortCL

`XYZSortCL` sorts the entries of two one-dimensional arrays into increasing order and applies the same permutations to the rows and columns of a second array.

```
void XYZSortCL (double *xp
                double *yp,
                double *zp,
                int nx,
                int ny,
                int ncols);
```

`xp, yp`	One-dimensional arrays containing the entries to be sorted.
`zp`	Two-dimensional array to be re-ordered.
`nx`	The number of elements in the array `xp`.
`ny`	The number of elements in the array `yp`.
`ncols`	The size of the second dimension in the declaration of the array `zp`.

InitCL

`InitCL` allows you to change the inital size of the main CCATSL window.

```
void InitCL (double left,
             double bottom,
             double top,
             double right);
```

`left, bottom`	The position of the bottom-left corner of the main CCATSL window, expressed relative to the bottom-left corner of the screen, as a proportion of its width and height.
`top, right`	The position of the top-right corner of the main CCATSL window, expressed relative to the bottom-left corner of the screen, as a proportion of its width and height.

InitStdioCL

`InitStdioCL` allows you to change the inital size and position of the Stdio (standard input/output) window.

```
void InitStdioCL (double left,
                  double bottom,
                  double top,
                  double right);
```

`left, bottom`	The position of the bottom-left corner of the Stdio window, expressed relative to the bottom-left corner of the main CCATSL window, as a proportion of its width and height.
`top, right`	The position of the top-right corner of the Stdio window, expressed relative to the bottom-left corner of the main CCATSL window, as a proportion of its width and height.

InitGraphicsCL

`InitGraphicsCL` allows you to change the inital size and position of the default graphics window.

```
void InitGrahicsCL (double left,
                    double bottom,
                    double top,
                    double right);
```

`left`, `bottom`	The position of the bottom-left corner of the default graphics window, expressed relative to the bottom-left corner of the main CCATSL window, as a proportion of its width and height.
`top`, `right`	The position of the top-right corner of the default graphics window, expressed relative to the bottom-left of the main CCATSL window, as a proportion of its width and height.

HaltCL

`HaltCL` terminates the program.

```
HaltCL();      /* Stop here */
```

Chapter 7

CCATSL variables, types and constants

7.1 Mathematical Functions

ODEFunctionCT

This type is used to declare user-defined functions passed to ODE solving routines such as `Rk4CL`. Such a function should take three arguments, a `double`, and two arrays of `double`s. See the example program demonstrating `Rk4CL` for an illustration.

7.2 Graphics

WindowCT

`WindowCT` is the CCATSL type used to referring to a window.

AxisModeCT

Arguments of type `AxisModeCT` control major aspects of how CCATSL draws graphs. Full details are in Section 3.3.2.

DrawDataCT (Standard plotting symbols)

Arguments of type `DrawDataCT` generally specify how points should appear in CCATSL graphics operations (though which values are sensible depend on the context). Whenever an argument of type `Drawdataoption` is required, one of `BLANK`, `DOT`, `PLUS`, `XCROSS`, `SQUARE`, `TRIANGLE` and `JOIN` is expected.

DrawObjectCT

Arguments of type `DrawObjectCT` control the appearance of complex plots such as contours and surfaces. `CONTOURS2D` requests a contour plot in the form of a flat ‘map’, `CONTOURS3D` produces a 3D plot, `WIREFRAME` asks for a wireframe surface plot, `SURFACE` produces a hidden line surface plot.

LineStyleCT

This type is used to define line styles. Possible values are `SOLID_LINE`, `DASH_LINE`, `DOT_LINE`, `DASHDOT_LINE`, `DASHDOTDOT_LINE`, and `NULL_LINE`.

BrushStyleCT

This type is used to define brush styles. Possible choicese are `SOLID_FILL`, `BDIAGONAL_HATCH`, `FDIAGONAL_HATCH`, `CROSS_HATCH`, `DIAGONAL_CROSS_HATCH`, `HORIZONTAL_HATCH`, `VERTICAL_HATCH`.

CursorStyleCT

This type is used to define cursor styles. Possible values are `ARROW`, `IBEAM`, `CROSS`, `RESIZE`, `WAIT`, `HIDDEN` and `VISIBLE`.

LineModeCT

This type is used to define the way lines should appear when they cross existing lines.

```
typedef enum {
  NORMAL_MODE, INVERT_MODE, EOR_MODE, OR_MODE, AND_MODE
} LineModeCT;
```

XYZRangesCT

This struture is returned by `IsotropicCL` to return the new ranges:

```
typedef struct
{
  double Xmin;
  double Xmax;
  double Ymin;
  double Ymax;
  double Zmin;
  double Zmax;
} XYZRangesCT;  /* values returned from IsotropicCL*/
```

WindowTypeCT

This type is used in conjunction with the window creation routines such as `WindowExCL`.

```
typedef enum {
  GRAPHICSWIN,      /* graphics window - mobile, fixed size, title*/
  SCROLLINGWIN,     /* scrolling graphics window - resizeable, title*/
  PLAINWIN,         /* plain window - fixed absolutely, no title*/
  TEXTWIN           /* text window - mobile, size box, title*/
} WindowTypeCT;
```

FontStyleCT

This type is used to specify desired font characteristics.

```
typedef enum {
  PLAIN, BOLD, ITALIC, BOLDITALIC
} FontStyleCT;
```

Colours

CCATSL uses the type `ColourCT` for colours, and defines a number of global variables of type `ColourCT` to give easy access to a range of common colours.

```
BlackCC,     BlueCC,         GreenCC,      CyanCC,
RedCC,       MagentaCC,      BrownCC,      GrayCC,
LightGrayCC, LightBlueCC,    LightGreenCC, LightCyanCC,
LIghtRedCC,  LightMagentaCC, YellowCC,     WhiteCC;
```

7.3 Miscellaneous

`ErrorFlagCD` and `ErrorMessageCD`

`ErrorFLagCL` is a global `int` variable used by the mathematical functions (Chapter 2) to signal an error. If an error occurs, `ErrorFlagCD` will be set to be `true` and the global `char*` variable `ErrorMessageCD` will point to a relevant error message. This may be displayed via `ErrorCL(ErrorMessageCD)`.

`boolean`

CCATSL defines the `boolean` type as `unsigned char`, and the constants `true` as 1, and `false` as 0.

Index