Learning to use C and the CATAM software library

CD Warner & AN Ross
Faculty of Mathematics, University of Cambridge

February 23, 2010

Contents

1 Introduction

This guide is intended to help you to learn how to program in C and how to use the CCATSL software library, whether you are new to programming, have programmed before but would like to know more about CCATSL and C, or are more experienced but need to look at some of the more advanced features of the CCATSL software library.

Skim over or skip sections containing material that is already familiar to you. However, you should realise that writing programs is the only way to learn a programming language — only by typing in and running your own programs will you learn to translate mathematics into computer programs. Although it might be tedious to type in long programs, instead of just loading them in, you can learn a lot in the process of typing in, running and changing the programs in this guide.

Sections 2–6 cover material suited to those who are new to Windows and C programming and mentions much detail that more advanced users may already know. These early sections concentrate on programming techniques by deliberately using examples that are mathematically very simple and by encouraging you to modify the example programs yourself.

Sections 7–10 cover material that is relevant both to those who are new to C programming and to those who have some programming experience. The material covered is producing plots of data using CCATSL library procedures and using C functions. Again the example programs are mathematically very simple and you are encouraged to modify them and write your own programs.

A short guide like this one can only cover a small subset of the CCATSL library and of the C language. CCATSL allows for the construction of menus to windows, making programs look professional and allowing the end-user to enter data using the mouse. This is covered in the last chapter of the CATAM Software library manual. You should consult reference books for information on some details. Pointers, `typedef`s and `struct`s are integral parts of C that you are likely to need in Part II, but which have not been covered in this guide. There are many books on C available. Two good introductory books are:
Practical C Programming, S Qualline, 1997,
and
C Programming Made Simple, C Sexton, 1997,

although there are many more and you should try and find one which suits you. Try looking in your college library. The original guide to the C language is:

The C Programming Language, 2nd Edition, BW Kernighan and DM Richie 1988;

this book is not a book for beginners to learn from, but is a useful reference.

The CCATSL manual (which is available in paperback from DAMTP Reception, at `http://www.maths.cam.ac.uk/undergrad/tripos/catam/ccatsl/` or in the CATAM Room at CMS) also contains much additional information that may be helpful to you. Finally, a number of example programs have been provided for you to run, copy and modify: see the directory C:\CCATSL\Examples (if you installed CCATSL on your own machine) for example programs.

2 Using Windows

2.1 Logging in

If the monitor screen is dark, check that the monitor is switched on and move the mouse around to switch off any active screensaver. If the computer is turned off (check the green power light is not lit) press the large round power switch. Wait until Windows XP has started and a window appears saying `Ctrl-Alt-Delete to Login or Shutdown`. While holding down the **Control** and **Alt** keys, press the **Delete** key. This will bring up a window which gives information relating to the `Computer Misuse Act 1990`. Move the mouse over the **OK** button and left click to close the window. A little window with a box for your user name and a box for your password will come up. Type in your user name and password in the right boxes and press **OK** with the left mouse button, i.e., move the mouse to point to point to **OK** and click the left mouse button . The computer will then log you onto the PWF. You will see a window appear with some startup messages but this will disappear when Windows has finished loading. A `Message of the Day` window will also appear. Press the red **X** button to close this window.

2.2 Windows basics

The Windows XP **desktop** contains several windows and icons. Each program (or **application**) running on the computer displays output in one or more windows. A window can be **minimised** so that it is hidden from view. This can stop the screen getting too cluttered.

Windows are controlled using the mouse or keyboard. These notes concentrate on mouse techniques which are easier for the beginner, but a few **keyboard shortcuts** are mentioned. Windows are controlled with the **left-hand** mouse button using one of the following operations:

- **Click:** Press and release the button without moving the mouse
- **Double click:** Click twice quickly
- **Drag:** Hold down the button and move the mouse

Because in earlier versions of Windows, many people found double clicking a nuisance, in Windows XP it is possible to set an option that permits a single click to do the same job; if this option is set on your computer, whenever this guide asks you to double click the mouse, a single mouse click will have the required effect. Click on a window to make it **active**. The active window appears in front of other windows and responds to characters typed on the keyboard. The active window also has a dark blue rather than a light blue **title bar**. A single click is also used to **select** an item prior to carrying out some action on it. A selection is marked by highlighting or a dotted rectangle.

A double click **chooses** an item. You choose an item to carry out an action, for example to start a program running from an icon on the dekstop.

Dragging is used to move or resize windows, select blocks of text for copying or deleting, etc.

2.3 The start menu and task bar

At the bottom of your screen you will typically see a blue bar with a **start** button and perhaps some other buttons on it. This is the **task bar**.

The green button on the far left with **start** on it is the **start menu**. It is used to start up other windows applications such as the CATAM software or Microsoft Word. Click on **start** and you will see a **menu** appear. The most important items are the ones at the top labeled **PWF Information** and **PWF Programs**. These contain nearly all the programs you will need. Click on **PWF Programs** and a window containing a number of **folders** appears. To find the CATAM software double click on the **Teaching Packages** folder then the **Catam** folder. Double click on the **C Programming Tools** icon to start. The **Catam News** icon gives useful information about the CATAM projects and the computer facilities and you should check it from time to time. The **Maths Computational Projects** icon takes you to the CATAM home page, from which you can access the electronic version of the project booklets and other useful information.

The task bar may also contain several other buttons, one for each window on the screen. If you click on a button the associated window will become the active window. In the far right of the task bar is a clock.

2.4 Window elements

Most windows have several elements which are used to control both the appearance of the window and its associated program.

- The **title bar** identifies the window. Drag the title bar around to move the window. The **active** window (the one receiving input, normally the foremost window), is shown with a bright blue title bar. **Click** anywhere on a window to make it active. Drag the **window border** to change its size.
- The **minimise button**, a blue button showing the _ character, hides the window from view. It still appears on the task bar. Pressing the associated button on the task bar will display the window again and make it the active window.
- The **maximise button**, a blue button showing a square, enlarges the window to its maximum size. For many applications the window fills the entire screen. The maximise button changes its appearance to show two overlapping rectangles. Click the button again to restore the window to its normal size.
- The **close button**, a red button showing a **X** character, closes the window and terminates the program if there are no other windows associated with it.
- A **scroll bar** (horizontal or vertical) is displayed when information such as a text document is too large to fit entirely in the window. Click in the scroll bar above or below the **scroll box** to move up or down a page in the document.

  Click the **scroll arrows** to move forwards or backwards through a document one line at a time. The position of the **scroll box** within the scroll bar indicates which part of a document is currently displayed in the window. Drag the scroll bar to move rapidly through a document.
- The window **menu bar** contains a list of menus of commands used to control the application. Select a menu by clicking its name, then choose a menu option by clicking or dragging. Most applications have a **File** menu which is used to save and load data files and a **Help** menu which gives information about the program.

2.5 Files and folders

My Computer and **Windows Explorer** are used to organise **files** and **folders** on the computer disks. Windows Explorer is usually found by clicking on **start**, then on **All Programs** then on **Accessories** then on the **Windows Explorer** icon. Information (C programs, data, results etc.) is stored in files which are given names consisting of two parts: a filename and (optionally) an extension. A filename must not contain any of the characters `\, /, :, *, ?, ", < or >` and should not contain any of `', (, )` or space. A filename extension consists of a full stop, followed by one to three letters or digits. The extension denotes the type of information stored in the

file. **.c** indicates that a file contains C program code. **.exe** indicates an executable program. It is not unusual for Windows XP to have been set up so that the extensions are not shown on the screen when using Windows Explorer.

The PWF PCs have several disk drives for storing information — a 3.5" drive (A:) which accepts removable floppy disks, and an internal drive (C:) which contains Windows. Additional "networked" drives are held on on fileserver computers. Drive U: holds your own files. I: holds the CATAM software.

When you are working with multiple files it is convenient to organise related files into groups called **folders** or directories. Applications access files by default in the current folder which can be changed using the application's File Menu. A file outside the current folder can be specified by a directory path preceding the file name:

I:\CATAM\ccatsl21\examples\a04curve.c

which refers to a file called **a04curve.c** in a subfolder of the **ccatsl21** folder called **examples**.

On-line help is accessible by clicking on the **start** button then on **Help and Support**. Regrettably, it can then often be a challenge to track down precisely what you need to know. One good strategy is to type some relevant words into the **Search** window and click on the white arrow in the green square may get you some helpful information. In particular, you may wish to search for instructions on creating folders, listing folders' contents, copying and deleting files, formatting floppy disks etc.

2.6 Logging out

When you have finished working you need to log off from the network to ensure that no-one else can alter or delete your files. First exit from any programs you have been running. To log out click on **start** then on the **Log Off** icon and finally on the the **Log Off** button.

double click on the **Logout from PWF** button on the left hand side of the desktop. After a few seconds you will see your programs being closed down and the little window saying `Press Ctrl+Alt+Delete to Login or Shutdown` appear again. You can now leave the computer.

3 C programming

3.1 CATAM C programming tools

We recommend that you use the **CATAM C programming tools** for the Maths Computational Projects. This contains the **Emacs** text editor, which is used to write and edit the programs, and which is integrated with the **gcc** compiler to actually compile and run the programs. This is explained in more detail later. The programming

tools also include the **C CATAM software library (CCATSL)** which contains numerous mathematical and graphical routines which simplify the writing of programs. This is available from the Maths faculty for use in the projects. It is pre-installed in the CATAM room in Mill Lane and in CMS, or you can download itfrom the CATAM web page and install it on your personal computer.

To start up the C programming tools select the item from the start menu as described in 2.3. When the emacs editor window appears you may see a window saying

`Starting emacs for CCATSL programming,` but most modern computers are so fast that this happens too quickly to see.

The emacs editor will load. It will display two windows. The first time you start CCATSL you will be taken through a short tutorial on using the software. In subsequent logins, the left window will display a list of the files in your project folder. The right window will display a menu providing help on the programming tools. To load a program into emacs, click on the name in the left window with the middle button. If your mouse only has two buttons then click them both at the same time. Try loading the example program **intro.c**.

3.2 Compiling and running a program

To run (or **execute**) a program it must first be translated or **compiled** into machine instructions. If the C compiler detects a **syntax error** in the program an error message and the line number where the error occurred are displayed. The error must be corrected before you can try to compile the program again. If the program is compiled successfully it is then linked with CCATSL routines and executed.

To execute the example program you first need to set `intro.c` as the **source file** which tells the compiler which program to compile. Select the **Catam** menu then the **Set source file** menu item. At the bottom of the window, you should see the line:

```
Specify source file name:  ~/ccatsl-projects/intro.c
```

where ~ indicates your home directory. If so, pressing **Return** will select `intro.c` as the source. If you see:

```
Specify source file name:  ~/ccatsl-projects/
```

type `intro.c` and then press **Return**. To compile the example program select **Catam** then the **Build target** menu item. You should see the left window split into two halves, the top half contains the source code while the bottom half contains the comments CCATSL makes while it compiles the source code. Later on, when we try to debug our programs, these messages will be useful.

Finally, to run the example program, select **Run program** from the **Catam** menu.

3.3 How to type in a program

This section is intended for the novice and should be skipped by those who already know a little about how Emacs and the C Programming Tools deal with opening, editing and saving files. In this manual, menu bar choices and menu option choices (that you select by clicking with the mouse) are shown in bold letters. For example:
Files then **Save Buffer**
means "click on **Files** in the menu bar then click on menu option **Save Buffer** in the menu that comes up".

1. Start up emacs as described in section 3.1.
2. List the current contents of your projects folder by selecting **Catam** then **List Project folder** (or simply press F5).
3. Create a new directory by clicking **Immediate** then **Create Directory**. At the bottom of the window, type `Tutorials` and press **Return**. This will create a sub directory where all the programs you write using this booklet will be stored. When you work on different CATAM projects, it is a good idea to create a different directory for each project.
4. Open this directory by clicking on `Tutorials` (which appears in blue) and then pressing **Return**.
5. Create a new file by clicking **Files** then **Open File** and then give the file a name by typing it in the line at the bottom of the emacs window. The name you type will be the name of the program and must end in `.c`. Our first program will be called displayname so type `displayname.c`.

All your files are saved on the U: disk (which is where your private files are kept) in a folder called `ccatsl-projects`. You will now see a blank window appear with the title **displayname.c** at the top of it and you can type in your own program. Try the following one which will be discussed in detail in section 4.

```
#include <catam.h>
int main(void)
{
  printf("Chris\n");
  return 0;
}
```

Once you are happy that you have correctly typed in your program, you can save it with **Files** then **Save Buffer** or — if you wish to change its name — **Files** then **Save Buffer As ...**. This is very useful if you want to write a new program that contains many of the lines you already wrote for an older program. Before running your program you have to compile it. This turns the program you write (which can be understood by humans) into something the computer can understand. As before, you can compile your program with

1. **Catam** then **Set source file** (or just hit the F6 button) making sure ~/ccatsl-projects/tutorials/displayname.c is the name of the source file at the bottom line of the window.

2. **Catam** then **Build target** (or just hit the F8 button)

3. Once it is compiled you can run the program with **Catam** then **Run program** (or just hit the F9 button.) This loads the program into the computer's memory and starts executing it at the `main()` function.

3.4 Finding the program you want to work on

To open a project you saved in an earlier session, click **Catam** then **List Project folders** and navigate through your files until you find <program name>.c, where <filename> means replace ¡filename¿ by the name of the file, and open it.

You will notice the item **Buffers** on the menu bar. Buffers in Emacs are documents that are being edited (like windows in Microsoft Word.) Emacs lists the names of the files on the left, and to the right (after a % sign), the location. If you have already opened <program name>.c, it should be somewhere on that list.

Finally, in addition to buffers, sometimes (especially after compiling, i.e., **Build**ing) your window will be split into two halves. To get rid of one of them, simply press F3 (or `Ctrl-x` and then 1).

3.5 Producing printouts

To print your program, select the window where your program is, and select **Catam** then **Print buffer**. Similarly, select the other CCATSL window, where the output of your program is, and again **Catam** then **Print buffer** to print the output of your program.

You will later see that CCATSL produces graphs as well. The quickest way to print graphs is to use Windows' in-built support for saving images using `Alt-PrintScreen`. Typing `Alt-PrintScreen` copies the content of the active window onto the clipboard. You can then paste it into any document (e.g., in Microsoft Word, select **Paste** from the **Edit** menu-bar option.) See section 10.4 for more advanced ways to save and print graphs.

3.6 Quitting Emacs

To exit from emacs click **Catam** then **Quit emacs**. If you have not saved all the files you have open then you will be asked if you want to save them. Press **y** for yes.

4 Displaying your name on the screen

Writing programs is the only way to learn a programming language. The program you typed in above in section 3.3 is one that simply displays my name on the screen (a standard first program for all beginners). Here is the program once again:

```
#include <catam.h>
int main(void)
{
  printf("Chris\n");
  return 0;
}
```

We will now explain each line individually.

1. A statement in C is a sequence of characters ending in a semi-colon ";".
2. The line beginning `#include` tells the compiler that the program could use anything in the CCATSL library or the standard maths and input/output functions.
3. The line `int main(void)` tells the compiler that this is the start of a section containing things to do. In C, all the bits of code that do something are contained in functions. You can think of functions as like building blocks for a program. Every program must have a function called `main()` which is where the program starts running. We will look at functions in more detail later on.
4. The line { tells the compiler that this is the start of the code contained in the function `main`
5. The line beginning `printf` displays anything that is in the brackets, in this case `Chris`, in a new window in the main emacs window. It doesn't display the `"` characters — the first `"` tells the compiler that a character string is beginning and the second `"` tells the compiler that the character string has finished. The characters `\n` are a special "control code" which means print a newline character. This moves the cursor on to the beginning of the next line.
6. The line `return 0;` tells the compiler that the function is finished and the program can exit.
7. Finally the line } tells the compiler that this is the end of the function.

Type this simple program into the computer and run it to check that it works. If you made a typing error (a very common one is to forget the ';' on one of the lines), the C compiler will provide a message in the emacs window to help you find your error and correct it. Now try a very simple change and alter the program to display your own name on the screen.

Unlike some other programming languages, the compiler differentiates between upper case and lower case letters. For example, as far as the compiler is concerned,

`Printf`, `PRINTF`, and `printf` are all completely different functions. Try changing the case of a few letters in the program to convince yourself that this is indeed true. Take care when typing in programs to get the case right!

5 Variables, assignments, expressions and operators

The next program writes out a table of the squares of the first 10 natural numbers. To do this several new concepts are introduced together, including defining and assigning values to variables, expressions to do mathematics, loops and formatted output. Variables are assigned values using the symbol `=` (often read as "set equal to".) The assignment statement takes the form

"variable" `=` "value"; where "value" can be a number or algebraic expression.

```
/*
  findsquare.c - This program prints out the squares of a set of
  consecutive natural numbers
*/
#include <catam.h>
int i;          /* number */
int isquare;    /* squared number */
int ilow;       /* smallest number */
int ihigh;      /* largest number */
int main(void)
{
  /* Set smallest and largest number */
  ilow = 1;
  ihigh = 10;
  /* Compute and display table */
  for (i=ilow ; i<=ihigh ; i=i+1)
    {
      isquare = i*i;
      printf("%4d  %4d\n",i,isquare);
    }
  return 0;
}
```

1. The lines

```
/*
  findsquare.c - This program prints out the squares of a set of
  consecutive natural numbers
*/
```

are a comment, which describes what the program is meant to be doing. A comment is any set of characters between `/*` and `*/` and is ignored by the compiler. Comments can appear anywhere in a line and are used to make the program clearer for reading by humans. Even if you wrote the program yourself, you will still find it easier to understand and debug if you comment it.

2. C requires each variable and the type of data it contains to be declared before use. Variables are defined at the beginning of a program and, as you will see later, at the beginning of a procedure or function. A declaration tells the compiler what types of variable the program will use and what their names are. For example:

```
int i;          /* number */
int isquare;    /* squared number */
int ilow;       /* smallest number */
int ihigh;      /* largest number */
```

   The text `int` tells the compiler that the variable following is an integer. All the variables in this program are integers. Alternatively, you could use `double` which is a standard real number that can be used with all the CCATSL library procedures. `double` quantities (which we shall use in section 6.3) are treated and stored differently in the computer memory than `int` quantities. C provides several other basic types — see the CCATSL manual for more details.

3. The variables `ilow` and `ihigh` are assigned values 1 and 10 respectively by the assignment statements:

```
ilow = 1;
ihigh = 10;
```

4. The syntax of the `for` loop is as follows. The loop starts with the word `for`. The next statement is the loop counter condition `(i=ilow ; i<=ihigh ; i=i+1)` which tells the computer to execute the loop once for each value of `i` from `ilow` to `ihigh`, adding 1 to `i` after each loop (i.e., once with `i` = 1 = `ilow`, once with `i` = 2 = `ilow` + 1, ..., once with `i` = 10 = `ihigh`). The computer sets `i` = `ilow` then it checks if `i` $\leq$ `ihigh` and if it is it executes the two statements enclosed in the { — } block. `i` is then incremented by 1, the computer rechecks if `i` $\leq$ `ihigh` and if true it repeats the two statements. This continues until `i` > `ihigh`. When `i` > `ihigh`, the loop ends and the computer carries on with the first line after the loop. Although the compiler does not care whether or not you leave blank lines or blank space between variables and operators, it is a considerable help when checking the logic of your program if you get into the habit of indenting the statements in loops. Decide on your policy for indentation and use of blanks and try to stick to it. You will find that when you type in a program the editor can automatically indent it for you.

5. The main work of the program is carried out in the two statements that form the loop. The assignment statement

```
isquare = i*i;
```

computes the square of `i` and assigns that value to the variable `isquare`. The "`*`" is an "operator" that means "multiply". Other operators include:
`/` (divide)[1]
`+` (add)
`-` (subtract)
See "Types" in the CCATSL manual for further information on variable types and operators.

6. The `printf` statement

```
printf("%4d  %4d\n",i,isquare);
```

displays the values of the variables `i` and `isquare` on the same line in a window (that automatically comes up on the screen). The string `"%4d  %4d\n"` tells the compiler the format in which to print the numbers. `"%4d"` tells the compiler that we want to print out an integer number with a width of 4 characters. There are two spaces printed between the two integers and after the second integer we want to print a newline. Specifying the width of the integers ensures the numbers are printed out in two neat columns.

Exercise:

Type the program into the computer and run it to check that it works. Then try some small changes to the program to modify what it does. For example you could

- Display a table of squares of the natural numbers 100–115
- Display a table with the squares of the odd numbers
- Display a table with `i, 2*i, 3*i` and `4*i`
- Make the tables more user friendly by adding a heading to the columns using some extra `printf` statements. Why should you not place these statements within the `for` loop?

6 Program flow control

The `for` structure is one of several control structures available for programming in C. In this section we will describe other control structures and invite you to change the

[1]Note that the meaning of divide depends on the nature of the variables; e.g., `1/2` takes the value `0` while `1.0/2.0` has the value `0.5`

program of section 5 to carry out the same simple task but using other control structures.

6.1 `while` loops

First consider the `while` loop. A very simple example which writes out the factors of a natural number is the program:

```
#include <catam.h>
int i;
int j;
int k;
int main(void)
{
  i = 360;   /* number to factor */
  j = 1;
  while (j <= i)
    {
      k=i/j;
      if(k*j == i)
        {
          printf("%4d",j);
        }
      j=j+1;
    }
  printf(" are factors of %4d\n",i);
  return 0;
}
```

The syntax of the `while` loop in this example is as follows. The loop starts with the word `while`. Then comes the loop test condition (`i<=10`) which takes the value `TRUE` if `i` is less than or equal to `10` and `FALSE` otherwise. Then comes a set of statements enclosed between the single indented { and }. If the loop test condition is `TRUE`, these statements are executed. The computer then re-evaluates the loop test condition, and if `TRUE` repeats the statements. This happens again and again until the loop test condition becomes `FALSE`; then the loop ends and the computer carries on with the first line after the loop.

The mathematically rather mysterious looking statement

```
j=j+1;
```

means "set the value of `j` to its old value plus 1". Can you work out how many times the computer will do the loop?

The statement

```
if(k*j == i)
```

is the first conditional test statement we have met — more on these later. If `k` multiplied by `j` is equal to (i.e., `==`) `i`, then the statements between the triple-indented { and } are executed. This is the case if `j` is a factor of `i`; `j` is printed to the screen. What value do you think `k` takes if `j` is not a factor?

Exercise:

Change the program in section 5 to use a `while` loop in place of the `for` loop. There are advantages and disadvantages in using each of these control structures. Try to think of an example that would be simpler to write using a `while` loop than a `for` loop. `for` loops are particularly useful when manipulating array and string variables (see section 8).

6.2 `do–while` loops

A very simple example which again writes out the first ten natural numbers is the program:

```
#include <catam.h>
int i;
int main(void)
{
  i = 1;
  do
    {
      printf("%10d\n",i);
      i = i+1;
    }
  while (i <= 10);
  return 0;
}
```

The syntax of the `do–while` loop is as follows. The loop starts with the word `do`. Then come two statements enclosed by { — } that are executed at least once even if the loop test condition is `FALSE`. Then comes the word `while` followed by the loop test condition (`i<=10`). If the loop test condition is `TRUE` then the computer repeats the two statements after `do`. When the loop test condition becomes `FALSE`, the loop ends and the computer continues with the first line after the loop. This is very similar to the `while` loop apart from the fact that the test condition is evaluated after executing the two statements rather than before. Sometimes one form is more convenient to use than the other. Can you think of an example?

Exercise:

Change the program in section 5 to use a `do–while` loop in the place of the `while` loop. Are there any situations you can think of where the `do–while` loop could not replace a `while` loop?

6.3 The `if–else` control structure

This control structure allows you to execute different sets of instructions depending on whether a test condition is `TRUE` or `FALSE`. As an example, let us consider the following program that uses the quadratic formula to solve a quadratic equation and gives the real or complex solutions as appropriate.

```
/*
  tryif.exe - Solve quadratic equation illustrating if-else
  statements.
*/
#include <catam.h> /* declarations of catam functions*/
double const tiny=1.0E-20;  /* Small non-zero number */
double a,b,c;          /* Coefficients of x*x, x and 1 */
double r,i;            /* Real & imaginary parts of solution */
double tmp1,tmp2;      /* Temporary real variables */
int main(void)
{
  a = 0;
  b = 0;
  c = 0;
  do
    {
      printf("Enter a, b and c  ");
      fflush(stdout);
      scanf("%lf %lf %lf",&a,&b,&c);
      if (fabs(a) <= tiny)
        {
          printf("Not a quadratic, a is too close to zero\n");
        }
    }
  while (fabs(a) <= tiny);
  tmp1 = b*b-4*a*c;
  if (tmp1 >= 0)
    {
      tmp1 = sqrt(tmp1)/(2*a);
      tmp2 = -b/(2*a);
      r = tmp2 + tmp1;
      printf("Solution 1 is %14.4f\n",r);
      r = tmp2 - tmp1;
```

```
      printf("Solution 2 is %14.4f\n",r);
    }
  else
    {
      r = -b/(2*a);
      i = sqrt(-tmp1)/(2*a);
      printf("Solution 1 is %14.4f + %14.4f i\n",r,i);
      printf("Solution 2 is %14.4f - %14.4f i\n",r,i);
    }
  return 0;
}
```

1. The program line
   `double const tiny=1.0E-20;  /* Small non-zero number */`

   is a statement called a constant declaration. Constants are declared at the beginning of the program and are set to a fixed value that cannot be changed. In this case we have declared a constant called `tiny` which is the magnitude of the smallest value of the variable `a` that the program considers to be non-zero: `1.0E-20` means 1.0×10^{-20}. Compare this with the line
   `double a,b,c;`

   which declares `a`, `b` and `c` to be real number variables. When the program is started they are each set to the value 0 by the first three lines of the `main()` function, but they can be changed later on.

2. The first `if` control structure does not have an associated `else` and is contained in a `do`–`while` loop. The computer repeats the instructions in the loop until the user has typed in a value of `a` with absolute value (`fabs(a)`) that is bigger than `tiny` (i.e., far enough from zero to be considered non-zero). If the absolute value of `a`, is less than or equal to `tiny` (too close to zero to be considered non-zero) then the line

   ```
   printf("Not a quadratic, a is too close to zero\n");
   ```

   is executed and the user is prompted to enter `a`, `b`, and `c` again.

3. Note that the statements

   ```
   printf("Enter a, b and c  ");
   fflush(stdout);
   scanf("%lf %lf %lf",&a,&b,&c);
   ```

   display the text `Enter a b and c` on the screen but do not move the output to the beginning of the next line. When writing on the screen the computer usually waits until it has several lines to print before displaying them to speed things up. The line `fflush(stdout);` forces the computer to print whatever it has waiting.

4. The line

```
scanf("%lf %lf %lf",&a,&b,&c);
```

will wait until the user types in three real numbers separated by spaces (not commas) before continuing with the next line. The `"%lf %lf %lf"` tells the computer that it is waiting for three real numbers. The three numbers are stored in the real variables `a`, `b`, and `c`. Note that the "Enter" key can be used instead of spaces when typing in the three numbers.

5. `tmp1` is initially set to `b*b-4*a*c`. The `if`–`else` control structure tests the value of `tmp1` to decide whether to output real or complex solutions. If `tmp1` is greater than or equal to zero, then the lines

```
tmp1 = sqrt(tmp1)/(2*a);
tmp2 = -b/(2*a);
r = tmp2 + tmp1;
printf("Solution 1 is %14.4f\n",r);
r = tmp2 - tmp1;
printf("Solution 2 is %14.4f\n",r);
```

   are executed. Following some simple mathematics, the real roots are written to the display. The syntax of the `printf` statement is slightly different to cope with the formatting of the `double` output. The format string `"%14.4f"` tells the computer to output the real variable in a field 14 characters wide with four decimal places of accuracy.

6. The expression `sqrt(tmp1)` computes the square root of `tmp1` (see section 9 for more details on functions).

7. If `tmp1` is negative then the lines

```
r = -b/(2*a);
i = sqrt(-tmp1)/(2*a);
printf("Solution 1 is %14.4f + %14.4f i\n",r,i);
printf("Solution 2 is %14.4f - %14.4f i\n",r,i);
```

   are executed instead and the complex roots are written to the display.

Exercise:

Type the program into the computer and run it to check that it works. Then try some changes to the program to modify what it does. For example you could

- Check if the two roots are the same and, if so, only output one of them. (Because the computer only stores real numbers approximately you will find that checking if a real number is equal to zero will not work — you will need to use a test that checks if a real number is "close" to zero).
- Write a much simpler program to find the square roots of a number, whether that number is positive or negative.

- Think of some examples to test some of the other logical operators i.e.,
  `==` equal to ($=$)
  `!=` not equal to ($\neq$)
  `>` greater than ($>$)
  `<` less than ($<$)
  `<=` less than or equal to ($\leq$)

6.4 The `switch-case-break` control

The `if`–`else` pair should be your default way of controlling the possible branching in execution. Sometimes though `switch-case-break` contributes to neater-looking code.

```
/* switch.c a switch-case-break example program */
#include<catam.h>
int option;
int main(void)
{
  do
    {
      printf("1:  Euler's \n");
      printf("2:  Leap Frog \n");
      printf("3:  RK \n\n");
      printf("4:  Quit\n\n");
      printf("Please enter your choice:");
      fflush(stdout);
      scanf("%d",&option);
      switch(option)
        {
        case 1:
          printf("Attempting Euler\n\n");
          /* Code to solve ODE using Euler */
          break;
        case 2:
          printf("Attempting Leap frog\n\n");
          /* Code to solve ODE using LF */
          break;
        case 3:
          printf("Attempting RK\n\n");
          /* Code to solve ODE using RK */
          break;
        }
    }
  while(option<4);
  return 0;
```

}

The most error prone aspect of `switch` is that at the end of each `case`, the command `break;` must appear. Otherwise, the program will "fall through" and execute the next case. Try leaving out one of the `break;` commands and watch with horror what happens.

7 `MainCL()` versus `main()`

We will now see how CCATSL can make the programs `switch.c` and `tryif.c` of the previous section more tidy.

1. Open `switch.c` and change `int main(void)` to `int MainCL(void)`.
2. Now compile the program. Using `MainCL` tells the compiler that we may use graphics in the program, and that we want graphical windows.
3. If asked by the compiler, select "Change program type to Window mode."

From now on we shall use `MainCL()` instead of `main()` when we write a program that only works in the CCATSL implementation of C. All functions which are specific to CCATSL end with `CL` and will not work with a standard C compiler. The advantage of using `MainCL()` is that it allows us to do everything `main()` allows, and more. As a default, you should use `MainCL()`. One thing to be aware of, when you now run the program, is that output sent to stdio (e.g. with `printf()`, but *not* `PrintfCL()`) does not appear in the buffer, but in a new **standard input/output window**. To print the content of the stdio window, select the option **Print Stdio data** from the control box (the Gnu icon in the top left corner) of the CCATSL window.

7.1 Entering data using CCATSL

When we wrote a program to find the solutions of a quadratic, we used `scanf()` to enter the coefficients `a`, `b` and `c`. Since the format of `scanf()` is not very friendly, CCATSL contains some user-friendly alternatives which can be used when `MainCL()` is used. To read an integer, we replace the three lines of code

```
printf("Please enter your choice:");
fflush(stdout);
scanf("%d",&option);
```

with the statement `option=ReadIntCL("Please enter your choice:",1);`. Not only do we not have to worry about forgetting the `&` in `&option` or if it should be %d or %lf, but we also get to set a default value!

8 Plotting graphs and sending output to a file

By the end of this section, you will have written a fairly long program which we now introduce in stages.

8.1 Arrays

We open the program `findsquare` of section 5, and save it as `displaysquares`. We now amend `displaysquares.c` as follows:

```
/*
  displaysquares.c - This program computes the squares of a set
  of consecutive natural numbers and plots a graph of the results
*/
#include <catam.h>
int i;          /* number */
double ri[10];  /* array of 10 real numbers */
double rsquare[10];  /* array of 10 real squared numbers */
WindowCT w1, w2;  /* ID of a Window Constant for output */
int MainCL(void)
{
  /* Define, open and prepare the window for output */
  w1 = WindowCL(0.1,0.1,0.8,0.8);
  WShowCL(w1);
  WTitleCL("Display demo");
  /* Compute and display table */
  for ( i=1 ; i<=10 ; i=i+1 )
    {
      ri[i-1] = i;
      rsquare[i-1] = i*i;
      PrintfCL(1,i,"%f",ri[i-1]);
      PrintfCL(16,i,"%f",rsquare[i-1]);
    }
  /*
    Next section, you should add the code to plot a graph here
  */
  return 0;
}
```

We again go through each new element in this program.

1. In the variables list, we no longer have `isquare`, but instead have two new variables, `rsquare` and `ri`,

   ```
   double ri[10];  /* array of 10 real numbers */
   double rsquare[10];  /* array of 10 real squared numbers */
   ```

   which are arrays of ten `double` quantities. An array is simply a matrix of values,

in this case a one dimensional matrix or vector. Eventually, the routine that plots the graph of `rsquare` against `ri` (see section 8.2) expects to plot `double` arrays and will fail if you try to plot `int` arrays.

2. The variable type `WindowCT` is CCATSL-specific, as are all types ending with `CT`, and colours ending with `CC`. They will not be recognised by other C compilers.

3. The `for` loop has also changed:

```
ri[i-1]=i;
rsquare[i-1]=i*i;
```

   These lines illustrate how to address the elements of the arrays. The index `i` of the array runs from 0 to 9 so `ri[i]` is the (`i`+1)th element of the `ri` array[2].

4. In a "Window mode" program using the `printf` function will display text in the standard input/output window. Instead we can create a dedicated window and display the text in that window. The lines

```
WindowCT w1, w2;
⋮
  w1 = WindowCL(0.1,0.1,0.8,0.8);
  WShowCL(w1);
  WTitleCL("Display demo");
```

   define a variable `w1` which is used to refer to the window, and then create and open the window on the screen. The four numbers in the `WindowCL` function set the size of the window as a proportion of the screen (the order being left, bottom, right, top.)

5. If you wish to play with fancy elements of the window (colours etc.) you will need to use the `WindowExCL()` function (see the CCATSL manual).

6. The `WShowCL` function actually draws the window on the screen and gives it the title given in `WTitleCL`.

7. The `PrintfCL` function is equivalent to `printf` except it has two initial arguments corresponding to the column and row where the text is to appear.

Exercise:

Type the program into the computer (or edit the program it was derived from) and run it to check that it works. Then try some small changes to the program to modify what it does. For example you could

- Change the `for` loop to create an array containing the first 10 factorials.

[2]Note that in this case the variable `ri[10]` is not defined and that an attempt to use it is likely to cause an error; unfortunately the compiler doesn't check for this sort of thing

- Add some code to compute the differences between consecutive squares.
- Change the `double` arrays to `int` arrays. What other things need to be changed to get the program to run? (Hint: don't forget `PrintfCL`)

8.2 Plots and windows

We will now add the necessary code to plot a curve in the default plotting window. At the end of the program, just before the final `return 0;` statement, we add the following lines of code:

```
PauseCL();
/* Plot ri against rsquare using defaults */
XYCurveCL(ri,rsquare,10,1,JOIN,BlueCC,AUTOAXES);
```

1. The `PauseCL()` procedure stops the program until you press a key and is a useful procedure to divide up the various parts of a program and allow you to produce paper copies of the graphical output between the pause procedures (see section 10.4).
2. The line

   ```
   XYCurveCL(ri,rsquare,10,1,JOIN,BlueCC,AUTOAXES);
   ```

   plots the array `rsquare` of real numbers against the array `ri` of real numbers. `ri` contains the x coordinates and `rsquare` contains the y coordinates of the graph which will be plotted. The next item in the list, (`10`), specifies that 10 values from the arrays should be plotted (i.e., all of them). The data points will be plotted joined (`JOIN`) by blue (`BlueCC`) lines and the axis ranges will be calculated automatically from the data (`AUTOAXES`). The CCATSL data plotting procedures all work with `double` data, hence the need to convert the `int`s we had in the original program into `double`s.

Add the extra lines to your existing program and run it to check that it works.

While plotting to the default plotting window is very straightforward, you will probably prefer to manually set up one or more plotting windows using CCATSL routines. We will now add the necessary code to define, open and clear a window and to plot the curve in that window. In CCATSL, a window is identified by a variable (of type `WindowCT`) that we must add to the variable list. At the end of the program, just before the `XYcurveCL` statement, we add the following lines of code:

```
/* Define and display window for graph */
w2 = WindowCL(0.5,0.1,0.9,0.9);
WShowCL(w2);
WClearCL();
```

```
WTitleCL("Plotting graph");
```

As before, the line

```
w2 = WindowCL(0.5,0.1,0.9,0.9);
```

defines a new window `w2`, a rectangle in the main "parent" (exe) window. Once the window is shown (and hence selected), the `WClearCL()` command clears the current window.

Exercise:

Add the extra lines to your existing program and run it to check that it works. Before modifying your program, save it in a separate file, as we will be using this program later on in section 10.1 when we plot two graphs on the same axes.

Now try some small changes to the program to modify what it does. For example you could

- Experiment with the size and colours of the window.
- Draw a different type of curve e.g. a quadratic.
- Superimpose two graphs in the same window (use `PRESET` in the place of `AUTOAXES` for the second graph).
- Add an extra window and plot something in that window too.

8.3 Writing output to a file

In this section, we will modify the program we created in sections 8.1 and 8.2 so that it writes output to a file instead of the screen. The new lines of code and modified old lines of code that are required to output the results to a file are as follows.

1. Define the output file by adding a variable `fv` of type `FILE` to the variable list:

   ```
   FILE *fv;       /* file variable */
   ```

2. At the beginning of the `MainCL` function open a file for output by adding the extra line of code:

   ```
   fv = fopen("moresqop.dat","w");
   ```

3. After the two `PrintfCL` statements, Add the `fprintf` below to write the results to the file. The `fprintf` statement works just like the `printf` statement but writes to a file instead of printing on the screen. The `fprintf` statement has `fv` at the beginning of the argument list to tell it which file to write to:

   ```
   fprintf(fv,"%14.4f  %14.4f\n",ri[i-1],rsquare[i-1]);
   ```

4. The `fprintf` actually writes into temporary memory. To tell CCATSL that we have nothing more to add to `moresqop.dat`, insert the line

   ```
   fclose(fv);
   ```

   just before the `return 0;` line at the end of the program. This transfers the data from the temporary memory to the file and closes it.

Exercise:

Type the program into the computer and run it to check that it works. Open the output file `moresqop.dat` to see the output. You can open a file in Emacs by selecting **Files** then **Open File...** from the menu bar. To practice working with output files you could

- Add headings to the table of squares in the output file.
- Change one of the programs from an earlier section to produce output to a file as well as to the screen.

9 Functions

Functions are self-contained structures contained in a C program that carry out particular tasks. They can take several variables as input and can return a value that can be used as part of an expression as though it were an ordinary variable. For example we could write a function `Ffact(i)` to compute the factorial of an integer variable `i`. It could be used to compute $2*3!+1$ (`result`) as follows:

```
result=2*Ffact(3)+1;
```

9.1 Functions not returning a value

Here is a typical function which does not return a value. It calculates the mean of an array of values. In some other languages a function which does not return a value is called a procedure. C does not differentiate between functions and procedures though.

```
void GetMean(int npts, double *xmean)
{
  int n;
  double xsum;
  xsum = 0;
  for (n=1;n<=npts;n++)
    {
      xsum += xdat[n];
```

```
    }
  *xmean = xsum/npts;
}
```

The layout is very similar to that of the simple programs in sections 4 and 5.

1. The line `void GetMean(int npts, double *xmean)` the function `GetMean` and the main (calling) program to share the variable. If the `*` is not included, as for `npts`, then only the value is passed and any changes made to the variable within the function will be lost when the function exits.

2. The lines

   ```
   int n;
   double xsum;
   ```

   define some variables. These variables exist only inside the function. The main program cannot access these variables. Also, if the main program were to contain an integer variable `n`, the computer would treat it as a different variable to the integer variable `n` in the procedure.

3. The statements

   ```
   {
     int n;
     double xsum;
     xsum = 0;
     for (n=1;n<=npts;n++)
       {
         xsum += xdat[n];
       }
     *xmean = xsum/npts;
   }
   ```

   do the work of the function. A one statement `for` loop computes the sum `xsum` of the array of values `xdat` (assumed to exist as a variable in the calling program). The next statement computes the mean of those values. The calling program can then use the value of `xmean` which has been calculated; but it cannot use `xsum`, which was only accessible within the procedure.

   Here is an example program that uses `GetMean`.

   ```
   /*
     average.c - Calulate the average of the squares of 1..50
   */
   #include <catam.h>     /* declarations of catam functions */
   int i,nvalues;
   double xbar;
   double xdat[100];
   ```

```
void GetMean(int npts, double *xmean)
{
  int n;
  double xsum;
  xsum = 0;
  for (n=1;n<=npts;n++)
    {
      xsum += xdat[n];
    }
  *xmean = xsum/npts;
}
int main(void)
{
  nvalues = 50;
  for (i=1;i<=nvalues;i++)
    {
      xdat[i] = i*i;
    }
  GetMean(nvalues,&xbar);
  printf("xmean = %14.4f\n",xbar);
  return 0;
}
```

1. After the `#include` statement, a list of **global** variables should follow. The global variables are `i, nvalues, xbar` and `xdat` and can be used anywhere in the program, by any function.

2. The 11 lines of the function `GetMean` should placed after the global variables section of the program and before the `main` function.

3. As explained, the variables defined in the function `GetMean` are called **local** variables and can only be used in that function.

4. The main program fills the `xdat` array with a sequence of square numbers. When the computer reaches the function call

   ```
   GetMean(nvalues,&xbar);
   ```

   the constant `npts` in the procedure is set equal to `nvalues` from the main program and the variable `xmean` in the procedure is set equal to `xbar` from the main program. The `&` before `xbar` means that we want to pass a pointer (the address) to `xbar`, not the value of the variable so that the main program can see the changes made by the function. Control transfers automatically to the beginning of the function `GetMean` and the statements in the function are executed.

5. When the computer reaches the final statement in the function, the value of `xbar` in the main program is equal to `xmean` in the function and control then returns to the statement immediately after the function call, i.e.,

```
printf("xmean = %14.4f\n",xbar);
```

which prints out the value of the mean.

Exercise:

Type in the program `average.c` and check that it runs. Then try some small changes to the program to modify what it does. For example you could

- Add code to the procedure to compute the standard deviation of the data and return it to the calling program.
- Remove `xmean` from the list of passed constants and variables and make it a variable in the main program instead.
- Write another function to read in values of `xdat` that you enter into the computer.
- Write another function to plot `xdat[i]` against `i` on the screen. (Hint: you will have to use `MainCL()` instead of `main()` to do this.)

9.2 Functions returning a value

Functions can also return a value directly to the calling program. A function can be referenced by including it within an expression as though it were an ordinary variable. For example:

```
double coshsq(double t)
{
  double cs;
  cs = (exp(t)+exp(-t))/2;
  return cs*cs;
}
```

This function computes the square of the hyperbolic cosine of a real variable `t` and returns it as `coshsq`. The `double` before the function name tells the compiler that the value returned by the function is a real number. Here is a very simple test program that produces a table of values of x, $\cosh^2(x)$, $\sinh^2(x)$ and $\cosh^2(x) - \sinh^2(x)$. It uses the function `coshsq` to calculate $\cosh^2(x)$ and calculates $\sinh^2(x)$ directly.

```
#include <catam.h>
double x,sinhsq,z;
double coshsq(double t)
{
  double cs;
  cs = (exp(t) + exp(-t))/2;
  return cs*cs;
```

```
}
int main(void)
{
  x = -2.0;
  while (x <= 2.01)
    {
      sinhsq = pow(((exp(x)-exp(-x))/2),2);
      printf("%6.2f  %6.2f  %6.2f  %6.2f\n",x,coshsq(x),sinhsq,coshsq(x)-
sinhsq);
      x += 0.2;
    }
  return 0;
}
```

In the function, the line `return cs*cs;` sets the return value of the function and in the main program the line

```
printf("%6.2f  %6.2f  %6.2f  %6.2f\n",x,coshsq(x),sinhsq,coshsq(x)-
sinhsq);
```

writes the value (`x`) of x, computes and writes the value (`coshsq(x)`) of $\cosh^2(x)$, writes the value (`sinhsq`) of $\sinh^2(x)$ and finally computes and writes the value (`coshsq(x)-sinhsq`) of $\cosh^2(x) - \sinh^2(x)$ for the current value of x.

Exercise:

Type in the program `hypertrig` and check that it runs. Then try some small changes to the program to modify what it does. For example you could

- Write a function to compute $\sin(x)/x$ and use it to plot the graph of $\sin(x)/x$
- Write a function `length(x,y)` to compute $\sqrt{x^2+y^2}$

9.3 Other predefined functions

This section contains a list of some of the basic C and CCATSL functions that are available for use together with a very brief description of their purpose. In the following list `r` and `s` are general `double` quantities and `i` is a general `int` quantity. The functions starting with a capital letter are CCATSL functions although they are used in exactly the same way as the basic C functions.

`fabs(r), abs(i)`	absolute value of quantity
`sqrt(r)`	square root of quantity
`sin(r), cos(r), tan(r)`	sine, cosine and tangent of quantity
`asin(r), acos(r)`	inverse sine and cosine of quantity
`atan(r)`	inverse tangent of quantity
`exp(r)`	exponential of quantity

`log(r)`	natural logarithm of quantity
`log10(r)`	log to the base 10 of quantity
`floor(r)`	largest integer $\leq r$ — returns `double`
`ceil(r)`	smallest integer $\geq r$ — returns `double`
`pow(r,s)`	computes r^s — returns `double`

You could investigate these functions further by writing a program to tabulate or plot some of them.

10 More advanced plotting

10.1 Two plots on the same graph with user defined ranges and axis labels

We start with a copy of `displaysquares.c` that we saved at the end of section 8.2. We will now add the necessary code to plot two graphs with user defined ranges and axis labels. Remove (or comment-out using `/* ... */` the line near the end of the program:
`XYCurveCL(ri,rsquare,10,1,JOIN,BlueCC,AUTOAXES);`

Instead, add the following lines of code:

```
/*
  Plot rsquare and ri against ri using
  user-specified axes and titles
*/
XRangeCL(-10,20);
YRangeCL(-100,150);
XAxisLabelCL("natural numbers");
YAxisLabelCL("square / natural");
/* Plot ri against rsquare using defaults */
XYCurveCL(ri,rsquare,10,1,JOIN,BlueCC,DRAWAXES);
XYCurveCL(ri,ri,10,1,JOIN,RedCC,PRESET);
```

1. The functions `XRangeCL` and `YRangeCL` define the ranges of the x and y axes to be plotted.

2. Similarly, `XAxisLabelCL` labels the x-axis and `YAxisLabelCL` the y-axis.

3. Most importantly, you must replace `AUTOAXES` with `DRAWAXES` so that the user-specified axis ranges will be used. If you forget, CCATSL will revert to automatic calculation of the ranges.

4. The second `XYCurveCL` procedure plots `ri` against itself (i.e., plots the curve $y = x$) in red (`RedCC`) using the pre-existing axes (`PRESET`).

Add the extra lines to your existing program and run it to check that it works. You could investigate other parameters associated with the graph (e.g. Axes' colours) using the CCATSL manual (section 3.3.2).

10.2 Using `CurveCL` to plot a function

The following example program is similar to our previous graph plotting programs, but computes squares using a function.

```
/* moresquares.exe - This program plots a graph of
  the function y = x*x   */
#include <catam.h>     /* declarations of catam functions */
int npts;          /* number of points to be plotted */
double rilow;      /* smallest number */
double rihigh;     /* largest number */
WindowCT w1, w2;  /* ID of windows for output */
double fun(double x)
{
  return x*x;
}
int MainCL(void)
{
  /* Define, open and prepare the window for output */
  w1 = WindowCL(0.1,0.3,0.5,0.9);
  WShowCL(w1);
  WTitleCL("PrintfCL demo");
  /* Set smallest and largest numbers */
  rilow = 1;
  rihigh = 10;
  npts = 200;
  PrintfCL(2,2,"Number of points is %d",npts);
  PauseCL();
  /* Define and display window for graph */
  w2 = WindowCL(0.5,0.1,0.9,0.9);
  WShowCL(w2);
  WClearCL();
  WTitleCL("CurveCL demo");
  /* Plot rsquare against ri using defaults */
  CurveCL(fun,rilow,rihigh,npts,BlueCC,AUTOAXES);
  return 0;
}
```

By now you should be familiar with most aspects of this program. The only new command is

`CurveCL(fun,rilow,rihigh,npts,CTblue,AUTOAXES);`

which plots the function `fun` for the range `rilow < x < rihigh`. `npts` data points will be plotted joined by blue (`BlueCC`) lines and the axis ranges will be calculated

automatically from the data (`AUTOAXES`). The simple function `fun` computes the square of its argument `x`.

Exercise:

Type the program into the computer and run it to check that it works. Then try some small changes to the program to modify what it does. For example you could

- Plot a different function with a different number of points and/or a different range of `x`.
- Plot more than one function on the same graph (see section 10.1.)

10.3 Plotting graphs point by point

Sometimes, we may wish to take full control on the way a graph is created. Here we demonstrate a short program to draw the numerical solution for an ODE, point by point.

```
/* euler.c a program to plot the numerical solution of an ODE */
#include<catam.h>
WindowCT w;
double derivative(double y, double t)
{
  return y*y;
}
double euler(int n, double t0, double h, double y0)
{
  int i;      /*counter*/
  double yn=y0;  /*current value of yn*/
  double tn=t0;  /*current time */
  for(i=1; i<=n; i=i+1)
    {
      XYDrawCL(tn,yn);
      yn=yn+h*derivative(yn,tn);
      tn=tn+h;
    }
  return yn;
}
int MainCL(void)
{
  int n;
  double t0;
  double y0;
  double h;
  y0=ReadDoubleCL("Enter initial value",1);
```

```
  t0=ReadDoubleCL("Enter initial time",0);
  h=ReadDoubleCL("Enter step size",0.01);
  n=ReadIntCL("Enter number of steps",20);
  w=WindowCL(0.1,0.1,0.8,0.8);
  WShowCL(w);
  XRangeCL(t0,t0+n*h*1.3);
  YRangeCL(y0-derivative(y0,t0)*n*h*5,y0+derivative(y0,t0)*n*h*5);
  GAxesCL();
  XYMoveCL(t0,y0);
  printf("t=%10.4lf, y=%10.4lf",t0+n*h,euler(n,t0,h,y0));
  return 0;
}
```

1. You first need to draw a graph by defining the `XRangeCL` and `YRangeCL` as we saw earlier in this section.
2. You then need to instruct CCATSL to draw the axes, using `GAxesCL`.
3. `XYMoveCL(x,y)` moves the pen to position (x, y) without drawing a physical line on the screen.
4. Conversely, `XYDrawCL(x,y)` will draw a straight line starting at the current position of the pen and ending at (x, y).

Exercise:

- The output of `euler.c` is currently messy with the standard input/output window obscuring the plot. How can you tidy it up a bit?
- Can you modify the program to solve an ODE using the Leap-frog method? the RK4 method?
- By consulting the CATAM Software Library manual, try to extend the Euler plotter to plot the solution to an ODE in two dimensions (Hint: you will need to use `XYZDrawCL` and `XYZMoveCL`).
- Can you combine these with the program `switch.c` to create a versatile ODE solver?

10.4 Producing printouts of plots using metafiles

This section will describe how to produce a metafile of a plot. A metafile is a small and quickly printable file of graphics instructions. Metafiles can be printed out or included in Word documents. You may recall from section 3.5 that alternatively, just hitting `Alt-PrintScreen` will save the active window onto the clip-board from where it can be pasted into Microsoft Word.

1. Run the program `displaysquares.c` (section 10.1).

2. When the program pauses just before drawing the two curves, click the mouse in the control menu box (the Gnu icon in the top left hand corner of the main parent (exe) window).

3. Select the **Record current window graphics** menu option and only then press a key to continue.

4. When the program pauses just before the final `return 0;` with the message in the menu bar: `... termination --- Press any key to quit`, click the mouse in the control menu box again and select the **Save recording as metafile** menu option. Then press any key to continue.

5. A standard **save-as** box appears where you are asked to give the file a name. Because of network timing issues it is unfortunately not possible to reliably save metafiles across the PWF network, i.e., to the U: disk. If you do try to save metafiles to U:, the computer is likely to crash and will need to be rebooted. Therefore, you should specify a local filename such as `C:\Documents and Settings\<metafile name>` or a floppy disk file such as `A:\<metafile name>`.

You can display and print metafiles directly from your programs. Alternatively you can insert them into a Microsoft Word document by clicking **Insert** then **Picture** and then choose **From file** and select the metafile you have just created.

10.5 Creating metafiles from within your program

Another way to produce a metafile is to include the instructions to start recording and saving a metafile within your code.

1. In `displaysquares.c`, just after the `PauseCL();` command, insert the statement `GRecordCL();`. This causes the start of a recording of a metafile.

2. Just before the `return 0;` statement, insert the line `GSaveCL("");`

The program will run and will prompt you for a name for the new metafile it just created.

Try producing plots and metafiles of other simple functions by editing the program. Try labelling the plot axes and superimposing two or more plots.