A Course in Scientific Modeling and Simulation

Mike O'Leary
Shiva Azadegan

Academx Publishing Services

Editor: Joann Manos
Cover Designer: Doris Bruey

Printed in the United States of America

ISBN: 1-932768-04-1

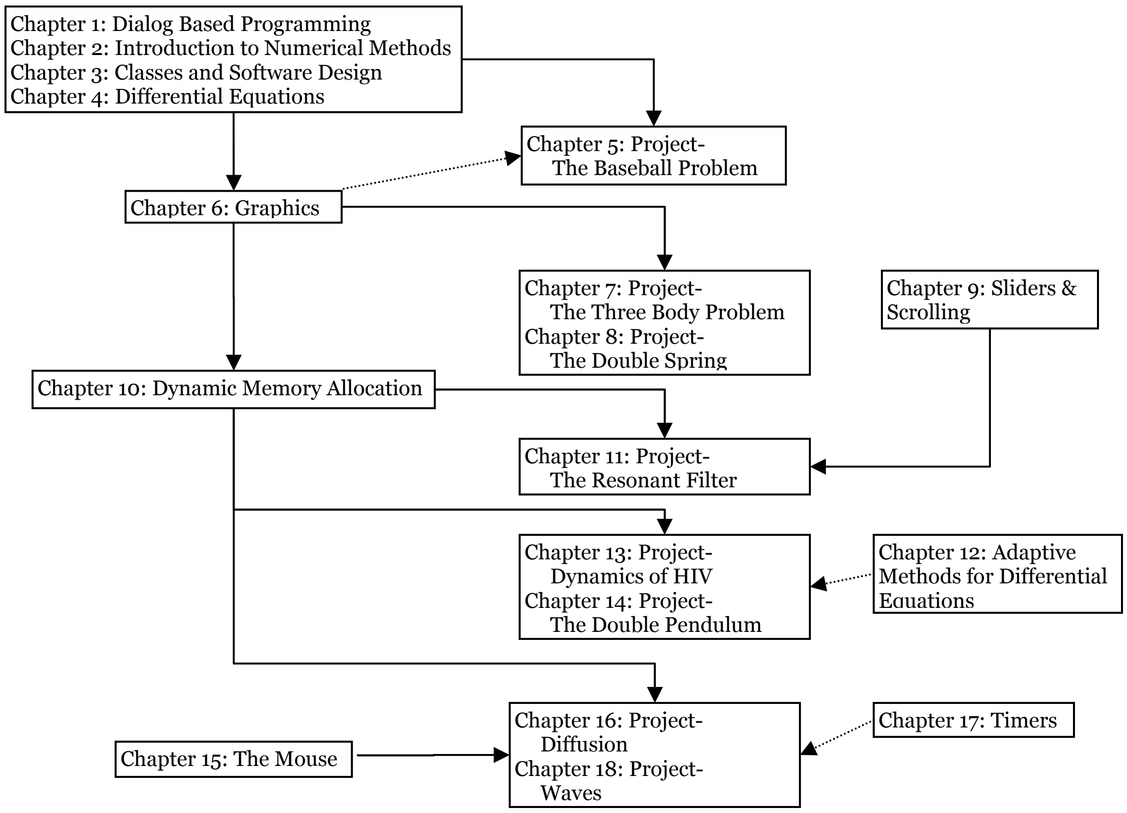
TABLE OF CONTENTS

Introduction

This book contains lecture notes and projects for our course *Scientific Modeling and Simulation*. This is an interdisciplinary course straddling the boundaries between mathematical modeling, numerical methods, and modern object-oriented computer programming. Our course is project-driven; we take realistic problems, model them, discuss appropriate numerical methods, and then create a simulation of the problem using Microsoft Visual C++ that takes full advantage of our computer's graphical capabilities.

This course was taught four times between 1999 and 2003. This book is designed as a reference for the instructor, though it can be adapted for use as a textbook.

Our course is a one semester course, and is organized around a sequence of three projects. Each time the course was taught we introduced some new projects, so this book actually contains eight projects. Thus, this book contains far more material than could be covered in a single semester. Further, not every topic was covered every semester; for example the material on timers (Chapter 17) was used only once. Below is a graph of the dependencies of one chapter on another; dependencies marked with a dashed arrow are optional. For example, adaptive methods are not required for the HIV dynamics project, but they can be


Chapter 1: Dialog Based Programming
Chapter 2: Introduction to Numerical Methods
Chapter 3: Classes and Software Design
Chapter 4: Differential Equations
Chapter 5: Project- The Baseball Problem
Chapter 6: Graphics
Chapter 7: Project- The Three Body Problem
Chapter 8: Project- The Double Spring
Chapter 9: Sliders & Scrolling
Chapter 10: Dynamic Memory Allocation
Chapter 11: Project- The Resonant Filter
Chapter 12: Adaptive Methods for Differential Equations
Chapter 13: Project- Dynamics of HIV
Chapter 14: Project- The Double Pendulum
Chapter 15: The Mouse
Chapter 16: Project- Diffusion
Chapter 17: Timers
Chapter 18: Project- Waves

used. Further, the baseball project can be given without first introducing graphics, or it can come afterwards, at the instructor's discretion.

The prerequisites for our course are Calculus 1 and 2, together with Introduction to Computer Science I, which is an introduction to programming in C++ up to classes.

The authors would like to thank the National Science Foundation, who supported this work through the Course, Curriculum, and Laboratory Improvement program, under grant DUE 9952625. We would also like to thank Dr. Teri Jo Murphy from the University of Oklahoma and Dr. Tad Watanabe from Penn State University for all of the work that they did to help us evaluate the effectiveness of our course. We would also like to thank Sam Houston who set up and maintained our lab for these past four years.

Dialog Based Programming

The Calculator

Section 1: Creating the Skeleton

In this section we describe how to use Microsoft Visual Studio to create a dialog based windows program that acts as a simple calculator. In particular, we shall use the MFC AppWizard to create a functioning skeleton program, to which we will add our own functionality.

We begin by creating the skeleton using the AppWizard. From Visual C++, select File, then New. You will obtain a dialog box like the one below.

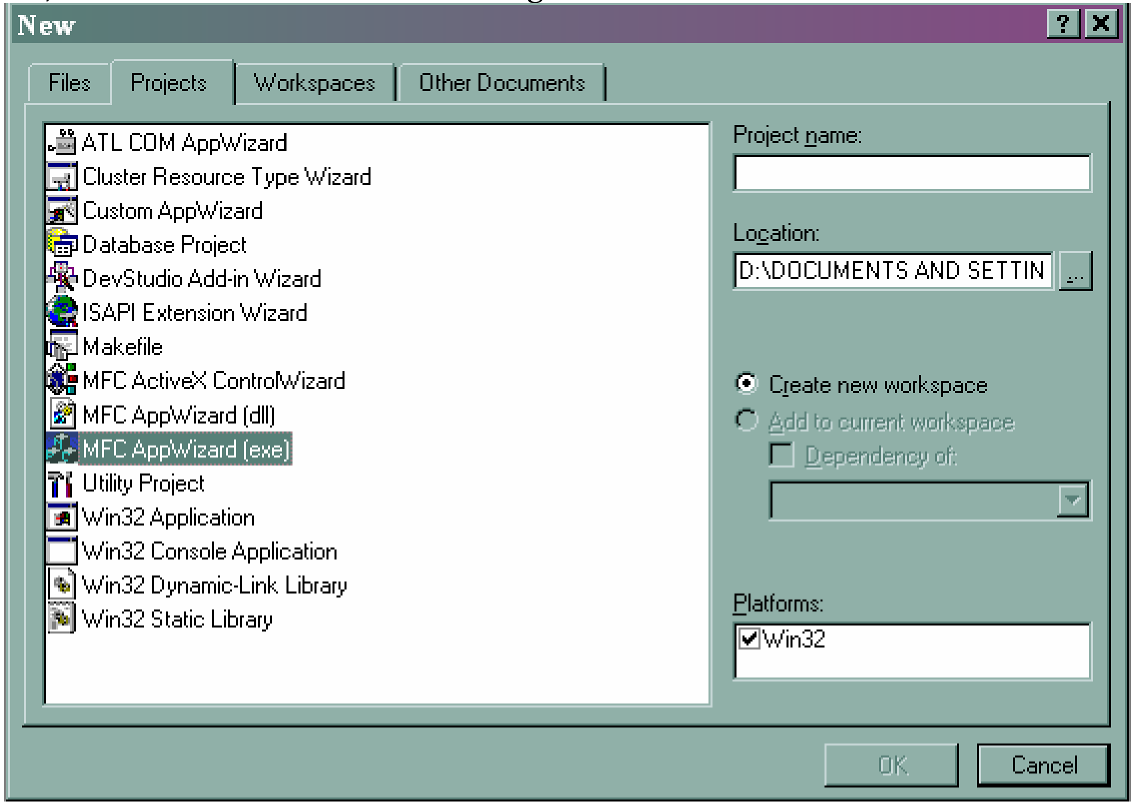


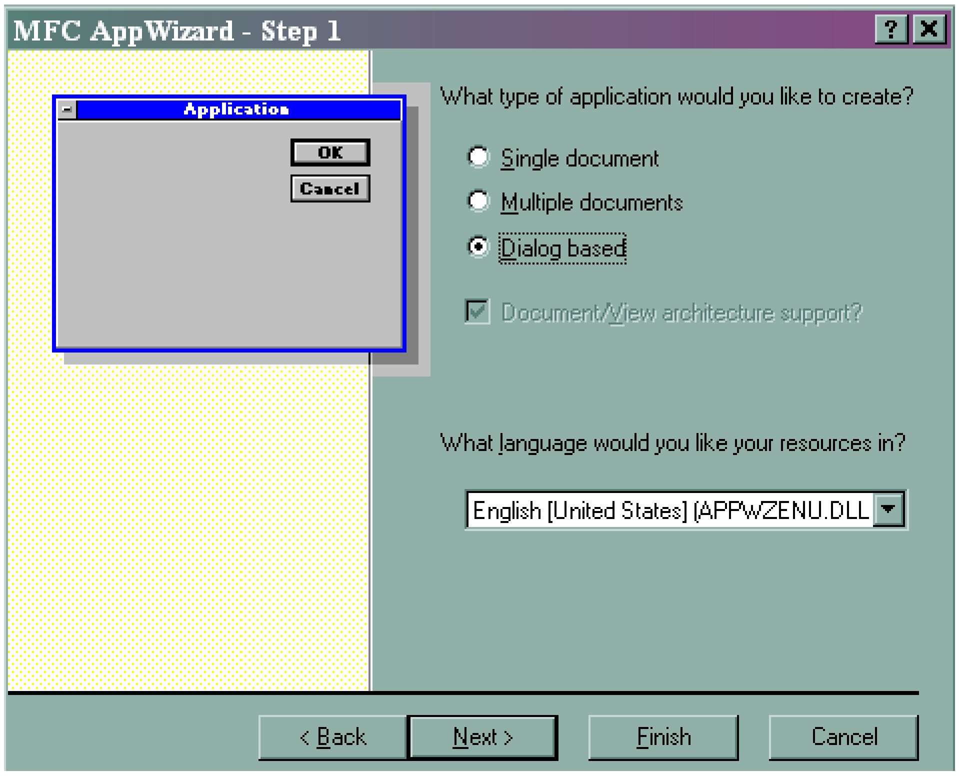
Figure 1: The MFC AppWizard

For this project, we select MFC AppWizard. Select a working directory and give the project a descriptive name. Note that, unless overridden, the project name will be used in the names of many of the classes that the AppWizard will generate. In our example, we shall call our project “Calculator”.

There are three options for an MFC AppWizard program-

- Single document
- Multiple document
- Dialog based

In these lecture notes, we shall focus exclusively on dialog based programs, in no small part because of their relative simplicity.


MFC AppWizard - Step 1

What type of application would you like to create?

- ( ) Single document
- ( ) Multiple documents
- [✓ Dialog based]

[✓] Document/View architecture support?

What language would you like your resources in?

English [United States] (APPWZENU.DLL

Figure 2: Select a dialog based program

In step 2, uncheck the box for the About box, and retain the remainder of the setting in their default state. Here you also have the option of changing the title of the main dialog box; however it is a simple matter to change this later. The default values in step 3 are suitable for our purposes, so leave those set as they are. Finally, in step 4 we see the names of the classes that will be constructed for us. Because our project is called Calculator, the classes are `CCalculatorDlg` and `CCalculatorApp`; we shall leave these names. Once we select Finish, the skeleton of our program will be created.

When this process is completed, you should be presented with a view like Figure 3. Before we continue working on the program, let us examine some of the features of Visual C++ and see how they can help us.

On the left side of the screen is a browsing tree with three tabs. The first tab shows the classes in the program, together with their associated variables and methods. Entries marked with are classes, so in Figure 3, we see the two classes `CCalculatorDlg` and `CCalculatorApp`. Entries marked are methods, while entries marked are variables. Methods or variables that are protected are indicated by , while private ones are indicated by . Double-clicking on the name of a class brings up the header file for that class; double-clicking on a variable brings up the header file at the point where that variable was declared; double-clicking on a method brings up the source file at the point where that method is implemented.

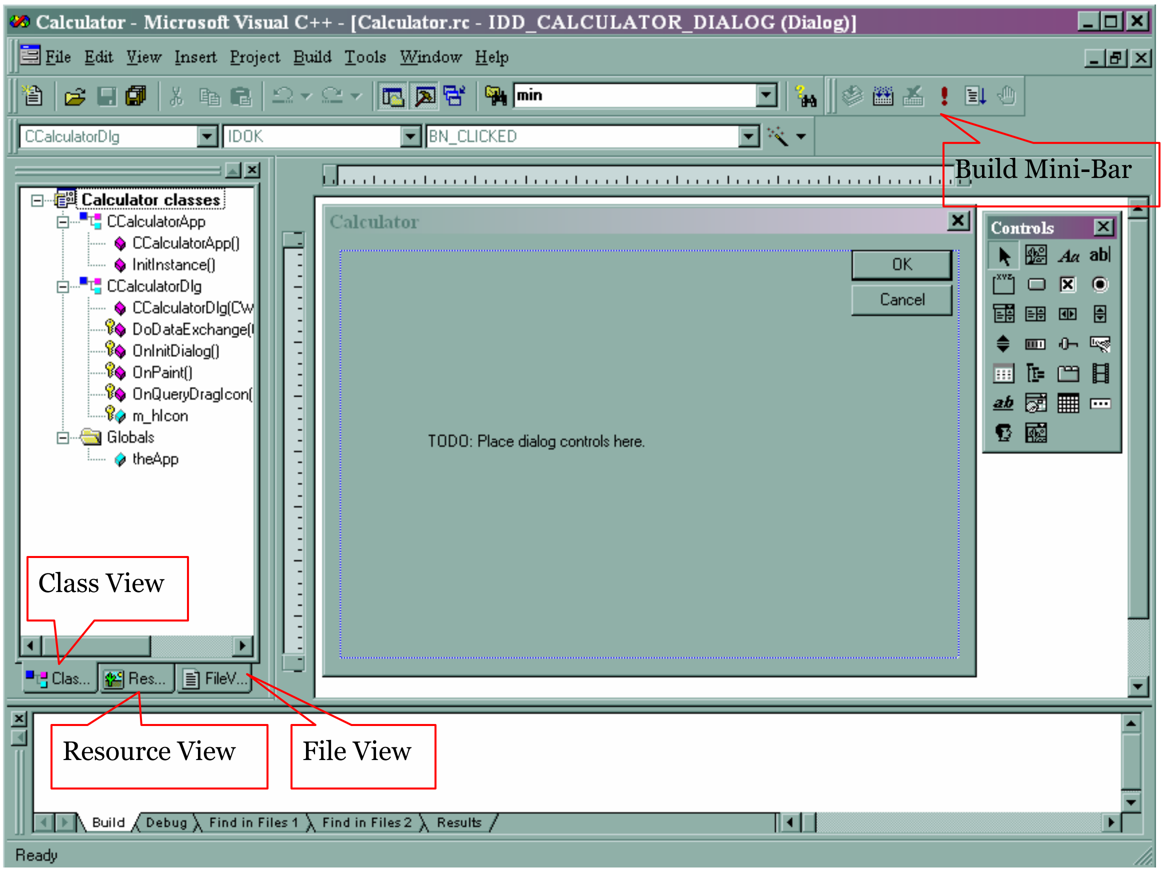


Figure 3: Our skeleton program

In the resource tab, we have a tree with all of the resources that the program possesses. Examples of resources are dialog boxes, icons, menus, toolbars, and string tables. Each resource has an associated ID, usually consisting of all capital letters. Our skeleton program has three resources- a dialog box `IDD_CALCULATOR_DIALOG`, an icon `IDR_MAINFRAME`, and version information `VS_VERSION_INFO`. We refer to these resources in our code by referring to the appropriate ID when needed.

The last tab lets us view the files that are used in the current project. The files are sorted by type- source files, header files, and resource files. Our program contains four source files, four header files, two resource files, and a readme.txt file. The last of these summarizes each of the files that the AppWizard has created for us.

The bulk of the screen is taken up by an editor. When editing source code, or header files, it acts as a text editor. When editing a dialog box, as shown in Figure 3, it acts as a graphical editor. The floating toolbar on the right in Figure 3 lets you add various components to the dialog box; we shall discuss this in more detail later.

Section 2: Compiling and Running the Program

Now that we have examined the structure that the AppWizard has provided for us, we would like to compile and run the program. To compile a source file, first open it for editing. Then, from the build menu select compile. You can also use the shortcut Ctrl+F7, or select the appropriate icon from the build mini-bar. To create the executable file, we need to build the application (F7); if the build process hangs, we can stop the build (Ctrl+Break). To execute the program, choose execute (Ctrl+F5). If the program has not been compiled and built, this command will tell you that files are out of date or do not exist; you can then build them before running the program. We debug our application by selecting Go (F5) and stop its execution at programmer defined breakpoints (F9).

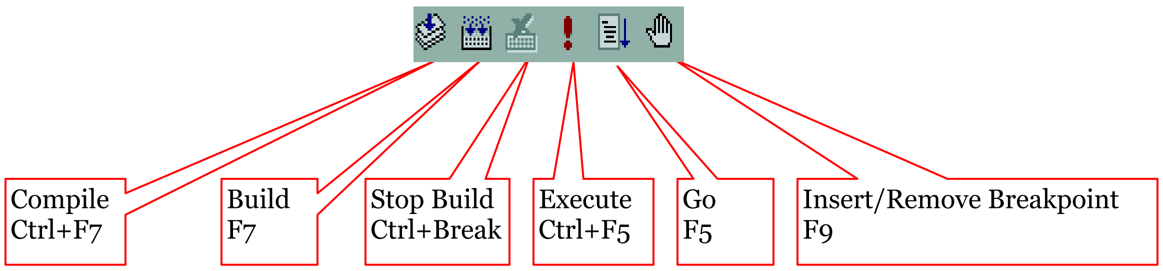


Figure 4: The build mini-bar

If we compile and run our skeleton program, we see that our program displays a dialog box with an OK button, a Cancel button, and the text "TODO: Place dialog controls here." The buttons work, but they simply cause the program to exit.

Section 3: Event Driven Programming

Now that our program skeleton has been created and is running, we can ask the question- How does our program actually function?

We notice that our program contains only one global variable- `theApp`, of type `CCalculatorApp`. This class is derived from the class `CWinApp` and contains the basic structure needed for a windows program. When the program runs, this object is created; it then runs the `InitInstance()` member function. The code for that function can be broken up into four distinct steps:

```
BOOL CCalculatorApp::InitInstance()
{
      AfxEnableControlContainer();
                                                      1
#ifdef _AFXDLL
      Enable3dControls();
#else
      Enable3dControlsStatic();
#endif
```

```
      CCalculatorDlg dlg;                                       (2)
      m_pMainWnd = &dlg;
```

```
      int nResponse = dlg.DoModal();
      if (nResponse == IDOK)
      {
            // TODO: Place code here to handle when the dialog is
            //  dismissed with OK
      }                                                         (3)
      else if (nResponse == IDCANCEL)
      {
            // TODO: Place code here to handle when the dialog is
            //  dismissed with Cancel
      }
```

```
      return FALSE;                                             (4)
}
```

The first step of the code is to handle basic initialization. More interesting is step 2, where an instance of the class `CCalculatorDlg` is created; a pointer to that instance is saved as `m_pMainWnd`. The class `CCalculatorDlg` is derived from the base class `CDialog`; one of the member functions of that class is called `DoModal()`. This method displays the dialog box until it is dismissed with either an OK signal or with a Cancel signal. Thus, in step 3 we see that the dialog box from `CCalculatorDlg` is displayed. It then waits until the dialog box sends an OK or a Cancel signal. In step 4, the `InitInstance()` function returns the value `FALSE`, and the program terminates.

Looking more closely at the code, one is left the following question- How does the program do anything? More specifically, once execution has passed to the first line in step 3, `int nResponse = dlg.DoModal();` shouldn't the program halt at that point and wait until the function call is resolved?

The explanation lies in the concept of *event-driven programming*. When an event occurs- a user presses a key, a program begins running, a mouse is moved (there are many others) Windows decides which program should handle that event. Windows starts running the appropriate code, and then waits for more events, without waiting for the code it started to terminate. In our case, the program runs up to the `dlg.DoModal()` command in step 3; however Windows will send other messages to our program that will be handled by other portions of the code. We shall illustrate this process in our calculator example.

Section 4: Editing the Dialog Box

To begin to put some function on our calculator program, let us start by modifying the basic dialog box created by our skeleton and shown in Figure 3. The floating Controls toolbar allows us to insert a number of different objects into our dialog box; we shall focus on three- static text, edit boxes, and buttons.

To construct our calculator, let us first select the existing static text "TODO: Place dialog controls here." and delete it. Next, add two edit boxes for the inputs, and one for the output. We also need static text boxes to label each of these. Finally we add buttons for each type of calculation- addition, subtraction, multiplication and division- that our program shall perform.

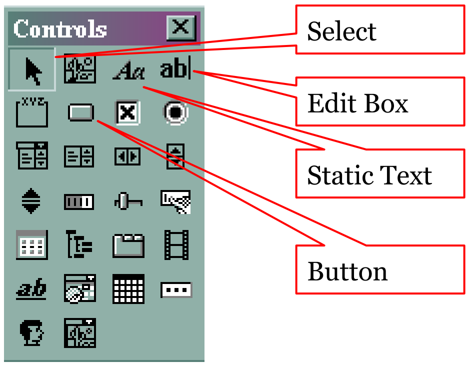


Figure 5: Controls

To add one of these items, simply click on the appropriate icon in the toolbar and then click on the place in the dialog box where you want the item to appear. To change the text that appears in a static text box or in a

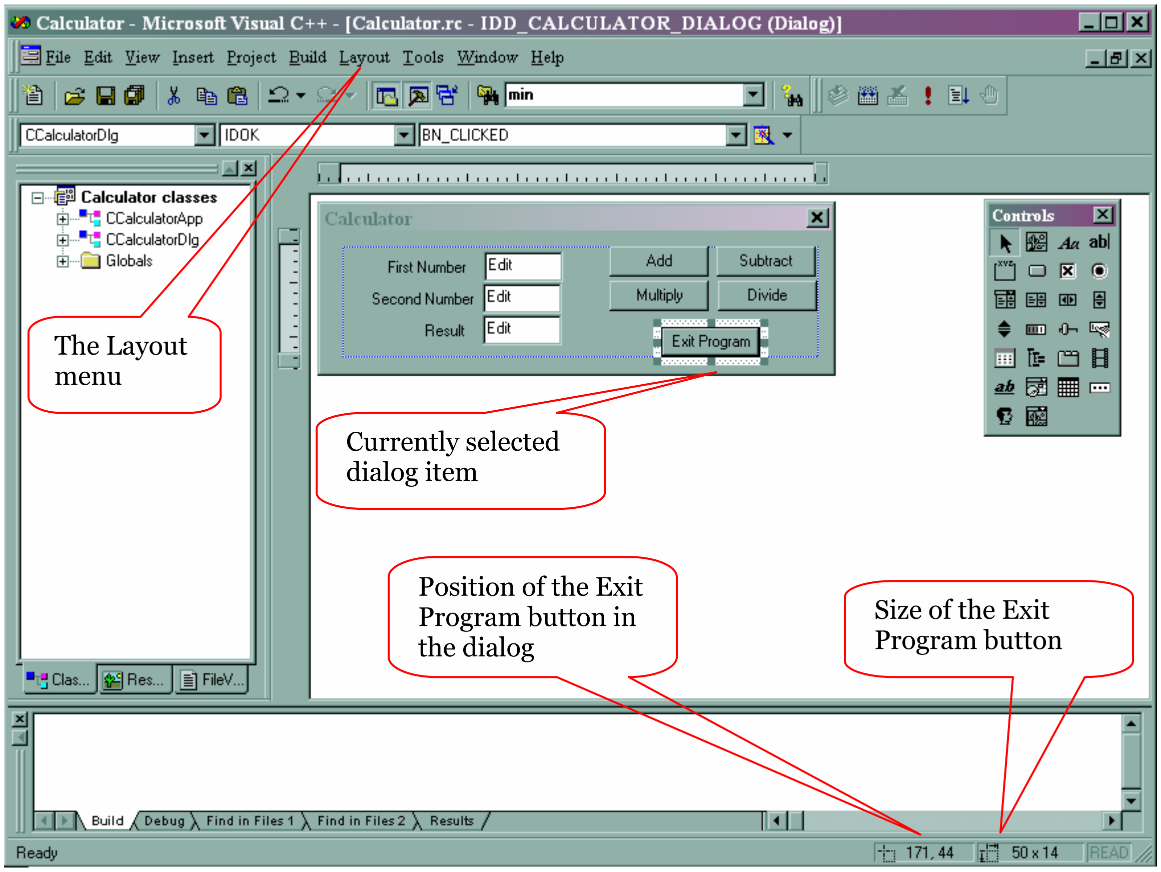


Figure 6: Editing our dialog box

button, simply select the object, and begin typing.

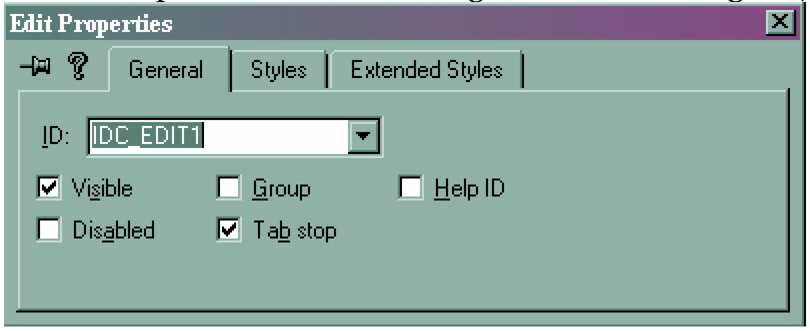
There are a number of tools available to simplify this process. On the main menu is an entry called Layout; from here you can align items, make them the same size, center them, and space them. When an item in a dialog box is selected, the bottom right corner of Visual C++ will display the size of the object, as well as its location in the dialog.

When we have finished, we should have a dialog box that looks something like the one in Figure 6.

Section 5: Adding Code

We would like to add some functionality to our program; we will do so by adding variables for the edit boxes we have created, and by writing code that will be executed when buttons are pressed.

We have three edit boxes to which we would like to associate variables. We begin the process by giving each edit box and each button a well-defined ID. Select the edit box corresponding to the first number in figure 6, right click, and select Properties. You will be presented with a dialog like the one in Figure 7.

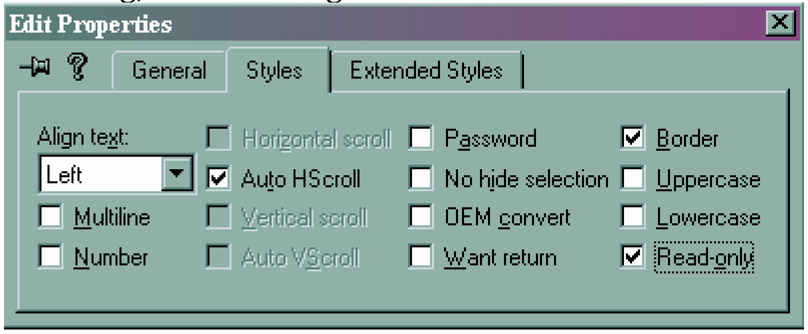


ID: IDC_EDIT1

✓ Visible
Group
Help ID
Disabled
✓ Tab stop

Figure 7: Editing the ID of a dialog item

We shall change the default ID to something more descriptive, like `IDC_FIRST_NUMBER`. We shall repeat the process with the second and third edit boxes, calling them `IDC_SECOND_NUMBER` and `IDC_RESULT`. Because the result box is to be used only to display the result and not for input, it should be marked read-only. We can do that by checking the Read-only box in the Styles tab of the Edit Properties dialog, as seen in Figure 8.



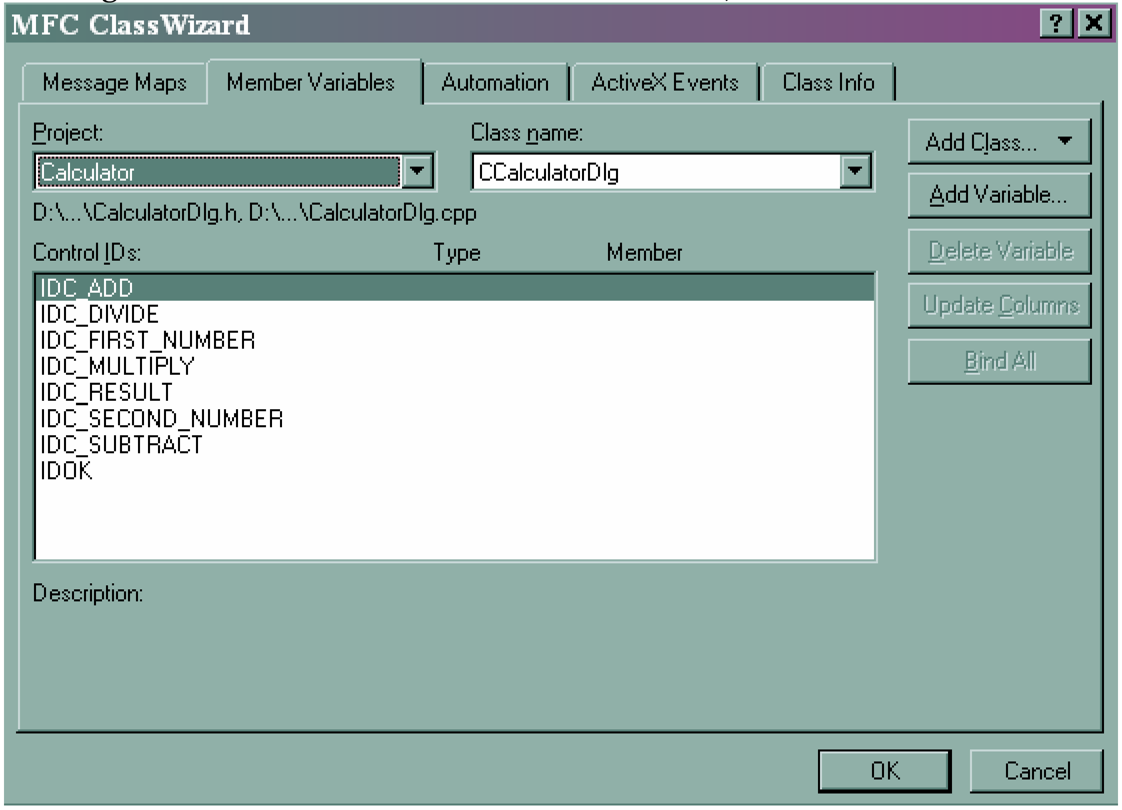
Align text: Left

Multiline
Number
Horizontal scroll
✓ Auto HScroll
Vertical scroll
Auto VScroll
Password
No hide selection
OEM convert
Want return
✓ Border
Uppercase
Lowercase
✓ Read-only

Figure 8: Selecting the Read-only box

We shall also change the ID of our buttons to `IDC_ADD`, `IDC_SUBTRACT`, `IDC_MULTIPLY` and `IDC_DIVIDE`.

Next, we shall associate variables with our edit boxes. We can do this by starting the Class Wizard. This is done by selecting Class Wizard from the View menu, using the shortcut key Ctrl+W, or from the various context menus that appear with a right click. Once the Class Wizard has started, select the Member



Project: Calculator
Class name: CCalculatorDlg
D:\...\CalculatorDlg.h, D:\...\CalculatorDlg.cpp

Control IDs:	Type	Member
IDC_ADD		
IDC_DIVIDE		
IDC_FIRST_NUMBER		
IDC_MULTIPLY		
IDC_RESULT		
IDC_SECOND_NUMBER		
IDC_SUBTRACT		
IDOK		

Description:

Figure 9: The Class Wizard- Member Variables tab

Variables tab; you will be presented with the dialog box in Figure 9. To add a variable, select a control ID, say `IDC_FIRST_NUMBER`, then select Add Variable. You will be presented with a box like the one in Figure 10. We created a double variable called `m_dFirstNumber` for that edit box; we proceed similarly creating doubles `m_dSecondNumber` and `m_dResult`.

A Note on Variable Names

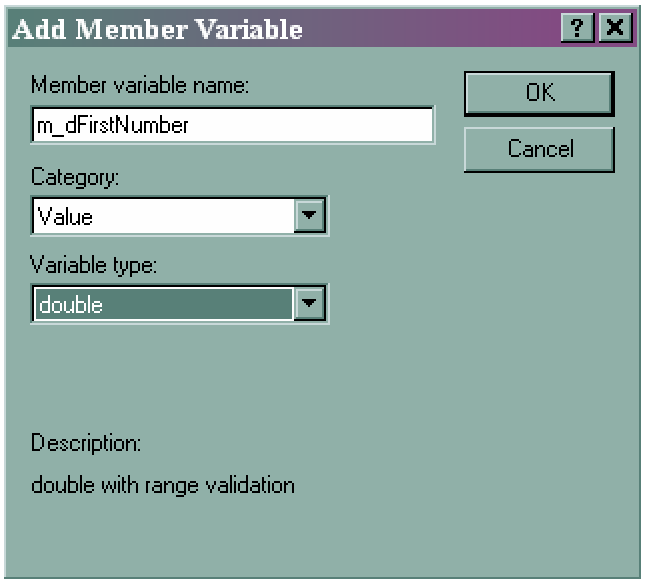
In our code, we shall always use Hungarian notation for our variables. These are a set of conventions used to make our code more readable. Names of classes start with `C`. If a variable is a member of a class, precede it with `m_`. Each variable has a prefix indicating its type

Prefix	Types
b	Boolean
cx	width

cy	height
d	double
dlg	dialog
i	int (integer)
n	short int
p	a pointer
rect	rectangle
s	string
wnd	window

The variable name is always capitalized. As examples, we have the class `CCalculatorApp`, and the member variable `m_dFirstNumber`. The advantage is that, simply by looking at a variable, we can tell immediately if it is a member variable, and its type.

Hungarian notation is not entirely standardized, and there are variations; the key is to be consistent in your code and understandable to others.


Add Member Variable

Member variable name: m_dFirstNumber

Category: Value

Variable type: double

Description:

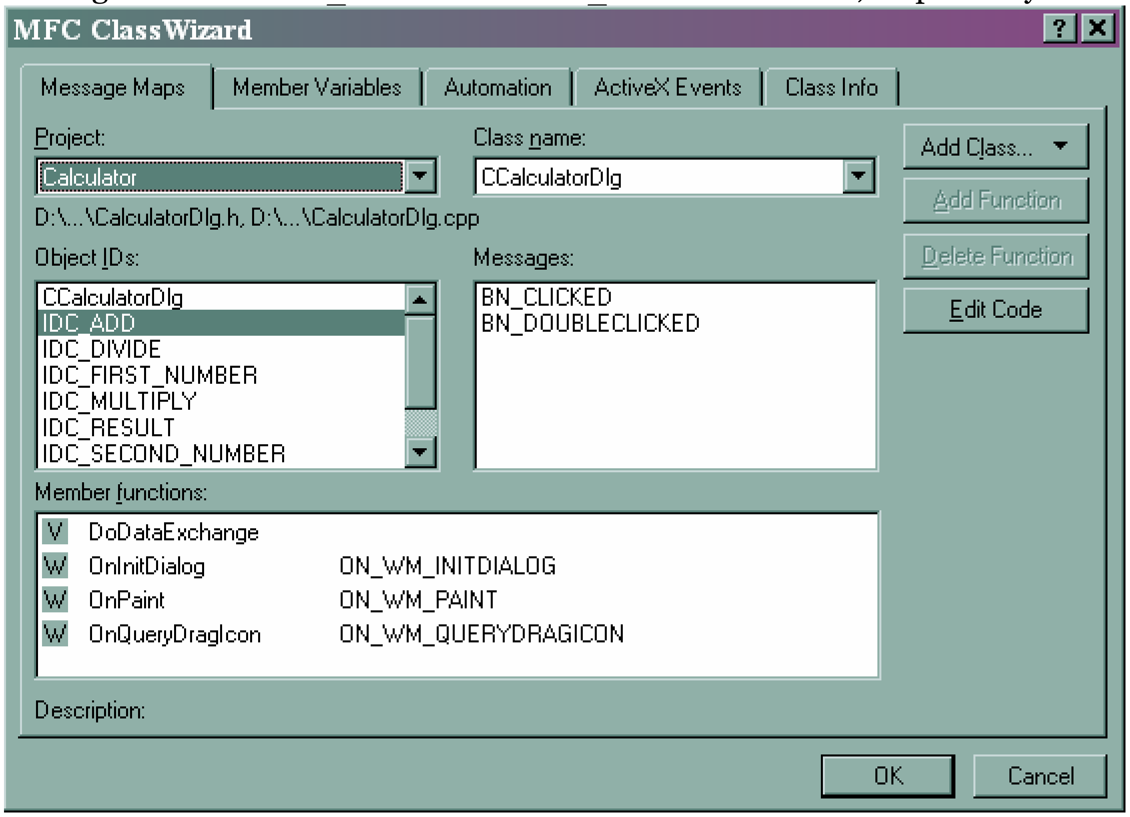
double with range validation

Figure 10: Adding a variable

Returning to our variables, note that variables assigned to edit boxes are public variables by default.

Next we would like to add some code that will be executed when buttons are pressed. We do this by assigning code to message maps. Whenever a button in a program is pressed, a message is sent to that program. We shall write the code that will execute when that message is received. We begin by starting the Class Wizard (Ctrl+W), and selecting the Message Maps tab. We are given a dialog box like that in figure 11.

Select the Object ID `IDC_ADD` corresponding to our Add button. This button can send two messages- one if it is clicked, and one if it double clicked; these messages are called `BN_CLICKED` and `BN_DOUBLECLICKED`, respectively.



Project: Calculator

Class name: CCalculatorDlg

D:\...\CalculatorDlg.h, D:\...\CalculatorDlg.cpp

Object IDs:

```
CCalculatorDlg
IDC_ADD
IDC_DIVIDE
IDC_FIRST_NUMBER
IDC_MULTIPLY
IDC_RESULT
IDC_SECOND_NUMBER
```

Messages:

```
BN_CLICKED
BN_DOUBLECLICKED
```

Member functions:

```
V  DoDataExchange
W  OnInitDialog      ON_WM_INITDIALOG
W  OnPaint           ON_WM_PAINT
W  OnQueryDragIcon   ON_WM_QUERYDRAGICON
```

Description:

Figure 11: Adding a message handler

To have code execute when the Add button is pressed, we simply select the `BN_CLICKED` message and select Add Function. We will then be prompted for a name of the resulting function; we can accept the default `OnAdd()`. When this is complete, we can select the Edit Code button to jump to the point in our source code for that function.

Our skeleton code is the following.

```
void CCalculatorDlg::OnAdd()
{
	// TODO: Add your control notification handler code here

}
```

We would like this code to take the values in `m_dFirstNumber` and `m_dSecondNumber`, add them, and store the result in `m_dResult`. However, simply adding the line of code

```
m_dResult = m_dFirstNumber + m_dSecondNumber;
```

to the `OnAdd()` function will not accomplish our goal. The reason for this is that, as currently constituted, the program does not take the text that the user has entered into the dialog box and convert it to values in our variables `m_dFirstNumber` and `m_dSecondNumber`; likewise, once the result is calculated, we have not told the program to actually display the result. The missing piece is a function called `UpdateData(BOOL)`. When `UpdateData(TRUE)` is called, the program will take the text in all of the edit boxes in the dialog, convert them, and store them in the appropriate variables in the class `CCalculatorDlg`. When `UpdateData(FALSE)` is called, the values in the variables in `CCalculatorDlg` are converted to text, and placed in the appropriate edit boxes in the dialog. Thus, our code for the `OnAdd()` function is

```
void CCalculatorDlg::OnAdd()
{
      // TODO: Add your control notification handler code here

      UpdateData(TRUE);
      m_dResult = m_dFirstNumber + m_dSecondNumber;
      UpdateData(FALSE);

}
```

With this beginning, it is simple to add the corresponding code for subtraction, multiplication and division.

One potential problem however, is what should the program do when the user asks to divide by zero? One way to alert the user is through the use of a message box. The function

```
MessageBox( LPCTSTR lpszText, LPCTSTR lpszCaption = NULL, UINT
nType = MB_OK )
```

takes three arguments, and can be called from any class derived from CDialog, like our CCalculatorDlg. The variable types for the arguments may be new; a LPCSTR is a pointer to a constant character string; for our purposes it is sufficient to know that we can use any piece of text enclosed in quotations as an argument. A UNIT is an unsigned integer; however we would not usually place the numerical value there. Instead there are predefined names, like MB_OK which we would use in its place.

The first argument is the text that will appear in our message box. The second is the caption of that message box, which by default is null. The last describes the icon that will appear; by default we use `MB_OK` which displays no icon. Other options are

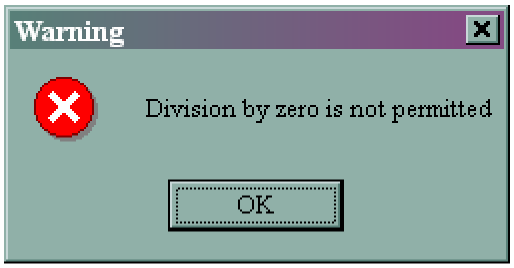
❌	MB_ICONERROR
❓	MB_ICONQUESTION
⚠️	MB_ICONWARNING

ℹ	`MB_ICONINFORMATION`

Thus, we can use the following code for our `OnDivide()` function

```
void CCalculatorDlg::OnDivide()
{
        // TODO: Add your control notification handler code here

        UpdateData(TRUE);

        if(m_dSecondNumber == 0)
                MessageBox("Division by zero is not
                        permitted","Warning",MB_ICONERROR);
        else
        {
                m_dResult = m_dFirstNumber / m_dSecondNumber;
                UpdateData(FALSE);
        }

}
```

Now, if a user asks to divide by zero, the following message is displayed.



Division by zero is not permitted

Figure 12: Result of our `MessageBox` command

Section 6: Saving and Loading

Your work is saved every time you compile your code, so you will rarely need to save it manually; if you do, be sure to use the “Save All” feature. Recall that your program consists of many different files; the “Save” feature saves only the file that you are currently using.

To load a program, it is simplest to use the “Open Workspace” command. This will load all of the files used for the program; the “Open” command will just open individual files.

To transfer your program from one computer to another, you need to copy all of the files in the root directory for your code, as well as all of the entries in the “res” subdirectory. You do not need to copy the contents of the “Debug” directory.

Section 7: Debugging the Code

A debugger lets the programmer step through parts or all of a program one line at a time. This lets us follow the variables and check that our code is acting as we think it should. To start the debugger, we can use the Go icon from the build min-bar (Figure 4) or press F5. If the program has been complied, it will execute until the code reaches a breakpoint. Then control passes to the programmer who can examine the values of the variables, and control how the program continues to execute. If no breakpoint is present in the code, the program executes normally.

To set or remove a breakpoint, select a point in the code, and use the insert breakpoint icon from the build mini-bar, or press F9. This type of breakpoint will trigger every time the debugger reaches this point of the code. There are times, however, when one wants to wait until some condition has been reached before a break; these advanced breakpoints can be inserted by selecting Breakpoints from the Edit menu, or by pressing Alt+F9. Once a breakpoint is created, you will see a red dot in the margin at that point.

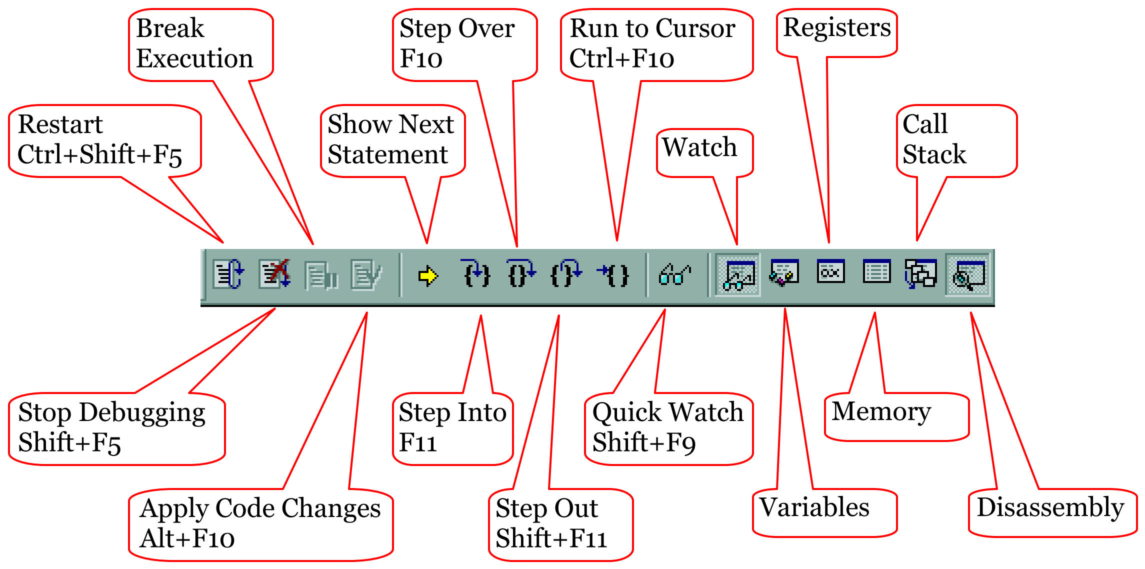


Figure 13: The Debug Toolbar

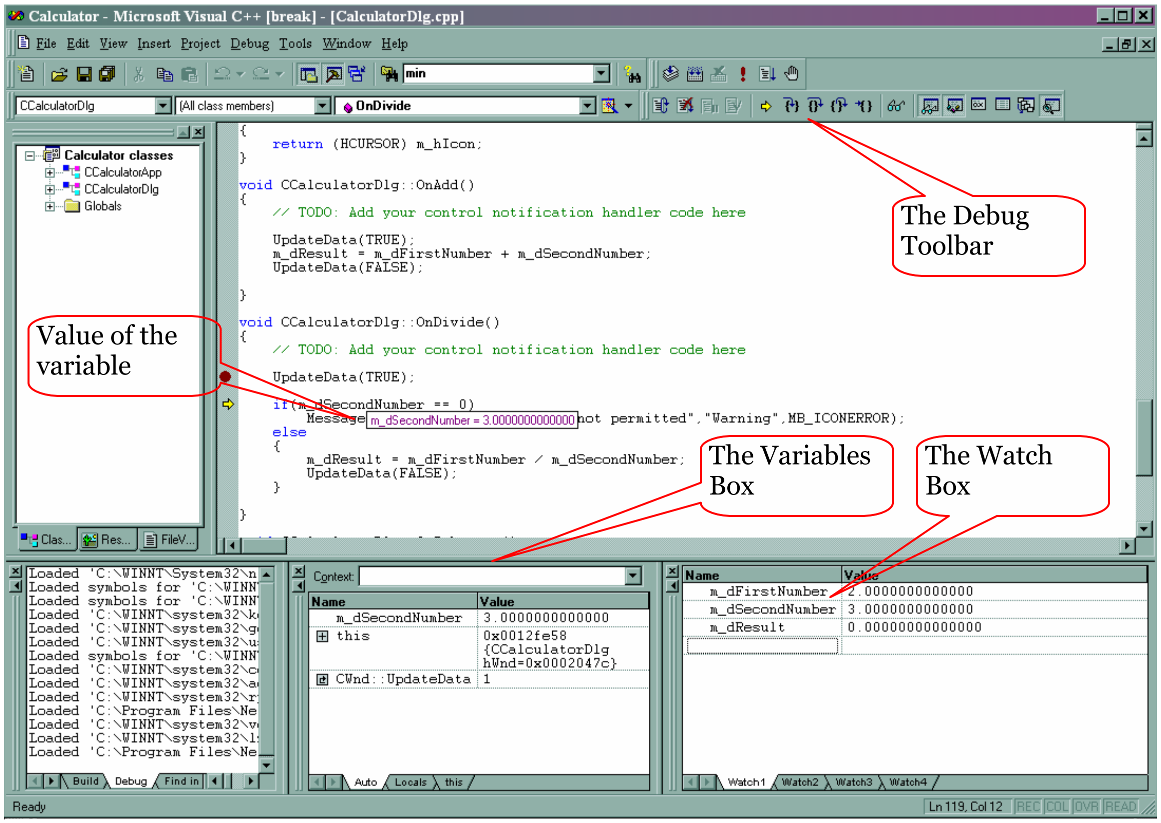
When a breakpoint is reached, the program will pause. To continue execution, we have a number of options.

- Selecting Go from the build mini-bar (Figure 4) will cause the program to continue executing until it reaches the next breakpoint.
- The Step Into (F11) command will execute the next statement, or, if that statement is a function call, it will jump to the first statement of that function.
- The Step Over (F10) command will execute the next statement; if that statement is a function call, it will evaluate the result of that function.

- The Step Out command (Shift+F11) finishes executing the current function call, and jumps to the line after that function was called.
- Run to Cursor (Ctrl+F10) begins executing statements, and stops when it reaches the current cursor position or a breakpoint.

When a program has been stopped by the debugger, a great deal of information about the status of the program is available. If you hold the cursor over a variable, the value of that variable will be displayed. The Variables Debug box, which can be displayed by selecting the Variables icon on the Debug Toolbar, shows the variables which the debugger thinks are important. The Watch Debug box, which is displayed by selecting the Watch icon on the Debug Toolbar, lets the user choose which variables to see. It has four tabs, so that different collections of variables can be maintained. Whenever one of the variables changes values, it is marked in red in the Watch box and the Variables box.

It is possible to edit your code while the debugger is running. You can even compile the changes, and have the program continue its execution from the stop at which you stopped, with the same variable values. To do so, use Apply Code Changes (Alt+F10) from the Debug Toolbar.



```
{
    return (HCURSOR) m_hIcon;
}

void CCalculatorDlg::OnAdd()
{
    // TODO: Add your control notification handler code here

    UpdateData(TRUE);
    m_dResult = m_dFirstNumber + m_dSecondNumber;
    UpdateData(FALSE);

}

void CCalculatorDlg::OnDivide()
{
    // TODO: Add your control notification handler code here

    UpdateData(TRUE);
    if(m_dSecondNumber == 0)
        Message m_dSecondNumber = 3.0000000000000 not permitted","Warning",MB_ICONERROR);
    else
    {
        m_dResult = m_dFirstNumber / m_dSecondNumber;
        UpdateData(FALSE);
    }

}
```

```
Loaded 'C:\WINNT\System32\n
Loaded symbols for 'C:\WINN
Loaded symbols for 'C:\WINN
Loaded 'C:\WINNT\system32\k
Loaded 'C:\WINNT\system32\g
Loaded 'C:\WINNT\system32\u
Loaded symbols for 'C:\WINN
Loaded 'C:\WINNT\system32\c
Loaded 'C:\WINNT\system32\a
Loaded 'C:\WINNT\system32\r
Loaded 'C:\Program Files\Ne
Loaded 'C:\WINNT\system32\v
Loaded 'C:\WINNT\system32\l
Loaded 'C:\Program Files\Ne
```

Name	Value
m_dSecondNumber	3.0000000000000
⊞ this	0x0012fe58 {CCalculatorDlg hWnd=0x0002047c}
CWnd::UpdateData	1

Name	Value
m_dFirstNumber	2.0000000000000
m_dSecondNumber	3.0000000000000
m_dResult	0.00000000000000

Figure 14: Debugging a program

Section 8: Getting Help

One can get help on any C++ keyword or MFC function by simply putting the cursor over the word, and pressing F1. This will bring up MSDN Library. If your word appears in more than one context, you will given a list of topics and asked to select one.

Resources

The calculator program described in these notes is available electronically.

Assignment

Write a program that takes two numbers, say x and y, and returns either

- The mean $\frac{x+y}{2}$,
- The geometric mean $\sqrt{xy}$, or
- The harmonic mean $\frac{1}{2}\frac{1}{\left(\frac{1}{x}+\frac{1}{y}\right)}$,

depending on the button selected. Basic error checking should be performed. Use the debugger as needed.

Introduction to Numerical Methods

Numerical Integration

Section 1: Theoretical Background

We would like to develop methods that can be used to solve realistic mathematical problems with the aid of a computer program. To do so, we need to learn some numerical methods. Most of these techniques are based upon Taylor's Theorem, which we learned in Calculus 2. We begin by reviewing some fundamental results from Calculus 1, then we prove Taylor's Theorem. Next, we discuss how to use numerical methods to evaluate definite integrals.

We begin with Rolle's Theorem.

Rolle's Theorem: Let $f(x)$ be continuous on $[a,b]$ and differentiable on (a,b). If $f(a)=f(b)$, then there exists a point ξ in the interval (a,b) so that $f'(\xi)=0$.

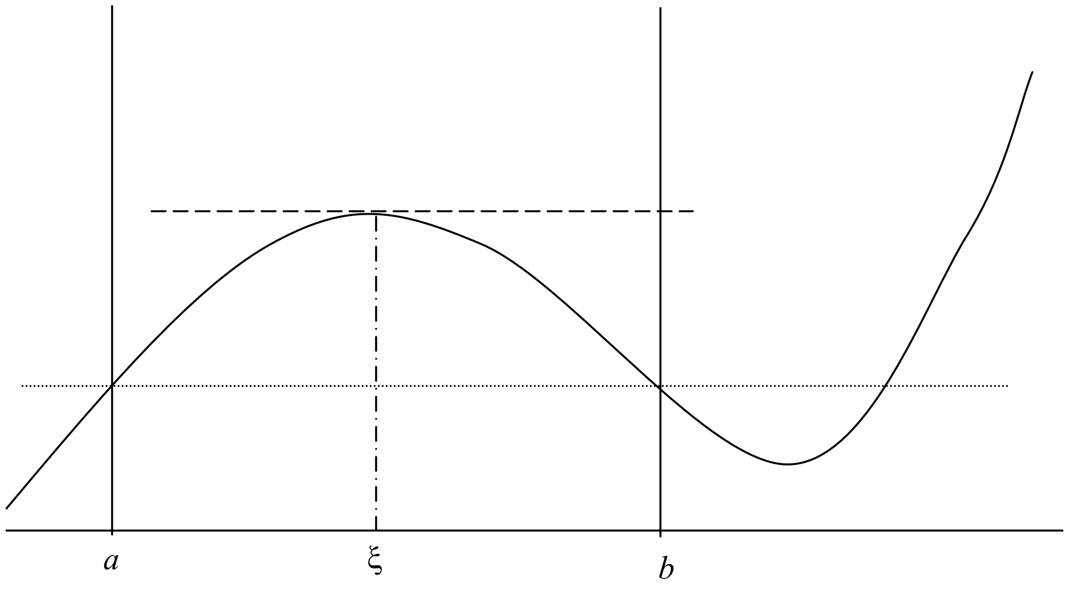


Figure 1: Illustration of Rolle's Theorem

Intuitively this says that if there are two points (say a and b) where the function has the same value, then there is a point between them (call it ξ) so that the derivative of f at that point is zero. There is no requirement that the point at which the derivative is zero is unique. For example, consider Figure 2, where we have moved the point b from Figure 1 to the right. Now there are two points, labeled ξ_1 and ξ_2, so that $f'(\xi_1) = f'(\xi_2) = 0$.

Rolle's Theorem was proven in Calculus 1; we shall not repeat the proof.

Our next major result is the Mean Value Theorem.

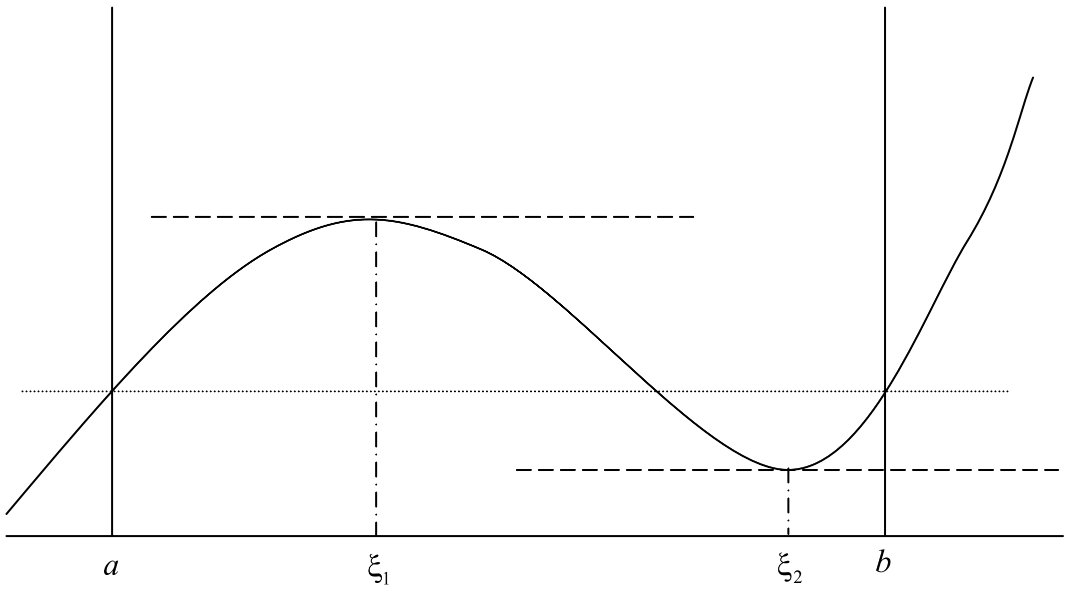


Figure 2: Second illustration of Rolle's Theorem

Mean Value Theorem: Let $f(x)$ be continuous on $[a,b]$ and differentiable on (a,b). Then there exists a point ξ in the interval (a,b) so that $f(b) - f(a) = f'(\xi)(b-a)$.

Intuitively, the Mean Value Theorem says that there is at least one point between a and b, say ξ, so that the tangent line to f through $(\xi, f(\xi))$ is parallel to the line through $(a, f(a))$ and $(b, f(b))$. For an illustration, see figure 3.

We can prove the Mean Value Theorem by appealing to Rolle's Theorem. Indeed, define the new function

$$\phi(x) = f(x) - \left[\frac{x-a}{b-a} f(b) + \frac{b-x}{b-a} f(a)\right]. \tag{1}$$

What is the function $\phi(x)$? It is the difference of the original function $f(x)$ and the equation of a line.

Note that if $x = a$, then

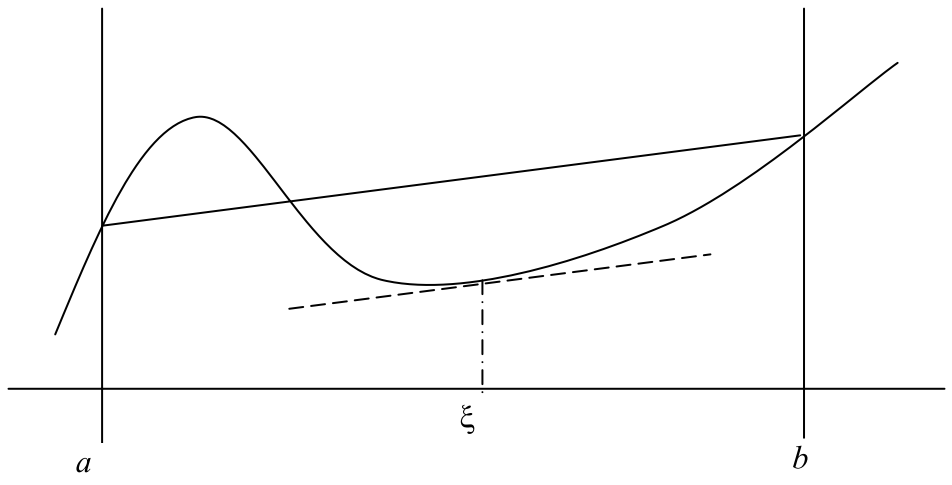


Figure 3: Illustration of Rolle's Theorem

$$\phi(a) = f(a) - \left[\underbrace{\frac{a-a}{b-a} f(b)}_{=0\cdot f(b)=0} + \underbrace{\frac{b-a}{b-a} f(a)}_{=1\cdot f(a)=f(a)} \right] = f(a) - f(a) = 0\,;$$

similarly

$$\phi(b) = f(b) - \left[\underbrace{\frac{b-a}{b-a} f(b)}_{=1\cdot f(b)=f(b)} + \underbrace{\frac{b-b}{b-a} f(a)}_{=0\cdot f(a)=0} \right] = f(b) - f(b) = 0\,.$$

Since $\phi(x)$ is the difference of $f(x)$ and a linear function, it is continuous on $[a,b]$ and differentiable on (a,b). Thus, Rolle's Theorem implies that there is at least one point, call it ξ in (a,b) so that $\phi'(\xi) = 0$. However, differentiating (1), we see that

$$\phi'(x) = f'(x) - \left[\frac{1}{b-a} f(b) + \frac{-1}{b-a} f(a) \right].$$

Since $\phi'(x)$ is zero when $x = \xi$, we see that

$$f'(\xi) = \frac{f(b) - f(a)}{b-a}$$

as required. □

One useful consequence of the Mean Value Theorem is that it gives us an estimate for the values of a function. Indeed, suppose that the value of f were known at some point, say x_0, and we would like to know the value of the function at some nearby point, say x. Setting $a = x_0$ and $b = x$ in the Mean Value Theorem, we find that

$$f(x) = f(x_0) + (x - x_0) f'(\xi) \tag{2}$$

for some unknown ξ between x_0 and x. Suppose all we knew about the function f was that $f(x_0) = 4$ and that $|f'| \le 1$. Then the Mean Value Theorem tells us that

$$4 - |x - x_0| \le f(x) \le 4 + |x - x_0|.$$

Thus we have been able to estimate the value of the function at the point x by using our information on $f(x_0)$ and on f'.

As estimates go, this is not terribly useful, because it simply uses the value of $f(x_0)$ to estimate the value of $f(x)$. However, there is a better result we can use to obtain these estimates: Taylor's Theorem.

Taylor's Theorem: Let $f(x)$ have $n+1$ derivatives on (a,b), and suppose that $f^{(n)}$ is continuous on $[a,b]$. For any x and x_0 in $[a,b]$, there exists a number ξ in (a,b) so that

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!} f''(x_0)(x - x_0)^2 + \frac{1}{3!} f'''(x_0)(x - x_0)^3 + \cdots + \frac{1}{n!} f^{(n)}(x_0)(x - x_0)^n + \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x - x_0)^{n+1}.$$

To understand this result, let us first see what it says if $n = 1$. In this case it says that given any x and x_0, there is a number ξ so that

$$f(x) = f(x_0) + (x - x_0) f'(\xi).$$

This is exactly what the Mean Value Theorem told us in (2) above. However, Taylor's Theorem goes even farther. It also says that

$$f(x) = f(x_0) + (x - x_0) f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(\xi) \tag{3}$$

and

$$f(x) = f(x_0) + (x - x_0) f'(x_0) + \frac{1}{2!}(x - x_0)^2 f''(x_0) + \frac{1}{3!}(x - x_0)^3 f'''(\xi) \tag{4}$$

for (probably different) values of ξ. The first of these is an approximation of $f(x)$ by a quadratic function, while the last is an approximation by a cubic. The power of Taylor's Theorem is that is lets us approximate an unknown function by a polynomial of arbitrary order, provided the derivatives exist and are continuous.

As an example, let us estimate the value of $\sin 40^\circ$. We begin by converting 40° to radians; we know that $40^\circ = \frac{2}{9}\pi$. We know the exact values of the trigonometric functions at $45^\circ = \frac{1}{4}\pi$, so we choose $f(x) = \sin x$, and set $x = \frac{2}{9}\pi$ and $x_0 = \frac{1}{4}\pi$ in (3). Thus

$$\sin\tfrac{2}{9}\pi = \sin\tfrac{1}{4}\pi + \left(\tfrac{2}{9}\pi - \tfrac{1}{4}\pi\right)\cos\tfrac{1}{4}\pi - \tfrac{1}{2}\left(\tfrac{2}{9}\pi - \tfrac{1}{4}\pi\right)^2 \sin\xi$$

for some unknown value of ξ between $\frac{2}{9}\pi$ and $\frac{1}{4}\pi$. Thus

$$\sin\tfrac{2}{9}\pi = \frac{\sqrt{2}}{2} - \left(\frac{1}{36}\pi\right)\frac{\sqrt{2}}{2} - \frac{1}{2}\left(\frac{1}{36}\pi\right)^2 \sin\xi .$$

Now we know that $\sin\xi$ is between -1 and 1. (Actually, we know a bit more- see the exercises!). Thus

$$\frac{\sqrt{2}}{2} - \left(\frac{1}{36}\pi\right)\frac{\sqrt{2}}{2} - \frac{1}{2}\left(\frac{1}{36}\pi\right)^2 \le \sin\tfrac{2}{9}\pi \le \frac{\sqrt{2}}{2} - \left(\frac{1}{36}\pi\right)\frac{\sqrt{2}}{2} + \frac{1}{2}\left(\frac{1}{36}\pi\right)^2 .$$

Evaluating these expressions, we see that

$$0.6415 \le \sin\tfrac{2}{9}\pi \le 0.6493 .$$

Thus we have an estimate for $\sin 40^\circ$ of about 0.64.

We can improve this estimate by using (4) instead of (3). In this case, we find that

$$\sin\tfrac{2}{9}\pi = \sin\tfrac{1}{4}\pi + \left(\tfrac{2}{9}\pi - \tfrac{1}{4}\pi\right)\cos\tfrac{1}{4}\pi - \tfrac{1}{2}\left(\tfrac{2}{9}\pi - \tfrac{1}{4}\pi\right)^2 \sin\tfrac{1}{4}\pi - \tfrac{1}{6}\left(\tfrac{2}{9}\pi - \tfrac{1}{4}\pi\right)^3 \cos\xi .$$

Thus

$$\sin\tfrac{2}{9}\pi = \frac{\sqrt{2}}{2} - \left(\frac{1}{36}\pi\right)\frac{\sqrt{2}}{2} - \frac{1}{2}\left(\frac{1}{36}\pi\right)^2 \frac{\sqrt{2}}{2} - \frac{1}{6}\left(\frac{1}{36}\pi\right)^3 \cos\xi .$$

Again, using the fact that $\cos\xi$ is between -1 and 1, we see that

$$\frac{\sqrt{2}}{2} - \left(\frac{1}{36}\pi\right)\frac{\sqrt{2}}{2} - \frac{1}{2}\left(\frac{1}{36}\pi\right)^2 \frac{\sqrt{2}}{2} - \frac{1}{6}\left(\frac{1}{36}\pi\right)^3$$

$$\le \sin\tfrac{2}{9}\pi \le \frac{\sqrt{2}}{2} - \left(\frac{1}{36}\pi\right)\frac{\sqrt{2}}{2} - \frac{1}{2}\left(\frac{1}{36}\pi\right)^2 \frac{\sqrt{2}}{2} + \frac{1}{6}\left(\frac{1}{36}\pi\right)^3 .$$

Evaluating these expressions, we find that

$$0.64259 \le \sin\tfrac{2}{9}\pi \le 0.64282$$

Thus we have a better estimate for $\sin 40^\circ$ of about 0.643.

Clearly we could continue this process indefinitely, obtaining better and better estimates for $\sin 40^\circ$. In fact, calculators use a variation of this process to evaluate trigonometric functions.

We would like to prove Taylor's Theorem, but to do so, we need an auxiliary result, called the Generalized Mean Value Theorem.

Generalized Mean Value Theorem: Let $f(x)$ and $g(x)$ be continuous on the interval $[a,b]$ and differentiable on the interval (a,b). Then there is a number ξ in the interval (a,b) so that

$$\left[f(b)-f(a)\right]g'(\xi)=\left[g(b)-g(a)\right]f'(\xi).$$

The proof of this result is similar in sprit to the proof of the Mean Value Theorem. We begin by constructing the auxiliary function

$$\phi(x)=f(x)\left[g(b)-g(a)\right]-g(x)\left[f(b)-f(a)\right]. \tag{5}$$

The hypotheses on f and g guarantee that ϕ is continuous on $[a,b]$ and differentiable on (a,b); thus there is a point ξ in (a,b) so that

$$\phi(b)-\phi(a)=\phi'(\xi)(b-a). \tag{6}$$

Differentiating the expression in (5), we find that

$$\phi'(x)=f'(x)\left[g(b)-g(a)\right]-g'(x)\left[f(b)-f(a)\right].$$

Thus, if we substitute into (6), we see that

$$\begin{aligned}&\left\{f(b)\left[g(b)-g(a)\right]-g(b)\left[f(b)-f(a)\right]\right\}\\ &\quad-\left\{f(a)\left[g(b)-g(a)\right]-g(a)\left[f(b)-f(a)\right]\right\}\\ &\qquad=\left\{f'(\xi)\left[g(b)-g(a)\right]-g'(\xi)\left[f(b)-f(a)\right]\right\}.\end{aligned}$$

Simplifying the left side, we find that it is zero and our result is proven. □

Proof of the Mean Value Theorem. With this preliminary result, we are now ready to prove the Mean Value Theorem. Let x and x_0 be fixed numbers. We introduce the new variable t, and define $g(t)=(t-x_0)^{n+1}$. We then introduce the auxiliary functions

$$F(t)=f(t)+f'(t)(x-t)+\frac{1}{2!}f''(t)(x-t)^2+\frac{1}{3!}f'''(t)(x-t)^3+\cdots+\frac{1}{n!}f^{(n)}(t)(x-t)^n$$

$$G(t)=g(t)+g'(t)(x-t)+\frac{1}{2!}g''(t)(x-t)^2+\frac{1}{3!}g'''(t)(x-t)^3+\cdots+\frac{1}{n!}g^{(n)}(t)(x-t)^n.$$

Apply the generalized Mean Value Theorem to these functions; thus there exists a value ξ so that

$$\left[F(x)-F(x_0)\right]G'(\xi)=\left[G(x)-G(x_0)\right]F'(\xi). \tag{6}$$

Our proof will be complete when we substitute back into this expression. We begin by calculating F' and G'. From the definition of $F(t)$ and the product rule, we find that

$$F'(t) = f'(t) + \left[-f'(t) + f''(t)(x-t)\right] + \frac{1}{2!}\left[-2f''(t)(x-t) + f'''(t)(x-t)^2\right]$$
$$+ \frac{1}{3!}\left[-3f'''(t)(x-t)^2 + f^{(4)}(t)(x-t)^3\right] + \cdots + \frac{1}{n!}\left[-nf^{(n)}(t)(x-t)^{n-1} + f^{(n+1)}(t)(x-t)^n\right].$$

Note that, because the derivative is being taken in t, the derivative of $(x-t)$ is -1, that the derivative of $(x-t)^2$ is $-2(x-t)$ and so on. (Chain Rule!) Examining our result, we see that the first two terms cancel out, as do the third and fourth. In fact, the only term that will remain is the last, so that

$$F'(t) = \frac{1}{n!} f^{(n+1)}(t)\cdot(x-t)^n .$$

Repeating the process for $G(t)$, we see that

$$G'(t) = \frac{1}{n!} g^{(n+1)}(t)\cdot(x-t)^n .$$

Because $g(t) = (t-x_0)^{n+1}$, we see that $g^{(n+1)}(t) = (n+1)!$ and thus

$$G'(t) = \frac{(n+1)!}{n!}(x-t)^n .$$

Next, we need to evaluate $G(x)$ and $G(x_0)$. The first is simple; indeed direct substitution shows us that $G(x) = g(x) = (x-x_0)^{n+1}$. On the other hand,

$$G(x_0) = g(x_0) + g'(x_0)(x-x_0) + \frac{1}{2!}g''(x_0)(x-x_0)^2$$
$$+ \frac{1}{3!}g'''(x_0)(x-x_0)^3 + \cdots + \frac{1}{n!}g^{(n)}(x_0)(x-x_0)^n .$$

Now $g(x) = (x-x_0)^{n+1}$, so

$$\begin{aligned} g(x_0) &= (x_0-x_0)^{n+1} = 0 \\ g'(x_0) &= (n+1)(x_0-x_0)^n = 0 \\ g''(x_0) &= (n+1)n(x_0-x_0)^{n-1} = 0 \\ &\vdots \\ g^{(n)}(x_0) &= (n+1)n(n-1)\cdots 1(x_0-x_0) = 0 \end{aligned}$$

so that $G(x_0) = 0$.

Substituting these values into (6), we find that

$$\left[F(x) - F(x_0)\right]\frac{(n+1)!}{n!}(x-\xi)^n = \left[(x-x_0)^{n+1} - 0\right]\frac{1}{n!}f^{(n+1)}(\xi)(x-\xi)^n$$

and hence

$$F(x) = F(x_0) + \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x - x_0)^{n+1}.$$

Because

$$F(t) = f(t) + f'(t)(x-t) + \frac{1}{2!} f''(t)(x-t)^2 + \frac{1}{3!} f'''(t)(x-t)^3 + \cdots + \frac{1}{n!} f^{(n)}(t)(x-t)^n,$$

we see that

$$F(x) = f(x)$$

and that

$$F(x_0) = f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2!} f''(x_0)(x-x_0)^2 + \frac{1}{3!} f'''(x_0)(x-x_0)^3 + \cdots + \frac{1}{n!} f^{(n)}(x_0)(x-x_0)^n.$$

Combining these, we obtain the statement

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2!} f''(x_0)(x-x_0)^2 + \frac{1}{3!} f'''(x_0)(x-x_0)^3 + \cdots + \frac{1}{n!} f^{(n)}(x_0)(x-x_0)^n + \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-x_0)^{n+1}$$

which completes the proof. □

Section 2: The Trapezoidal Rule

Our goal in this chapter is to learn how to approximate the values of definite integrals. Indeed, suppose we want to evaluate the definite integral $\int_{x_0}^{x_1} f(x)\ dx$ using a computer, where $x_0 < x_1$, and $f(x)$ is some smooth function. One approach is to approximate $f(x)$ be a straight line, and find the area beneath the resulting trapezoid; this is called the (simple) trapezoidal rule.

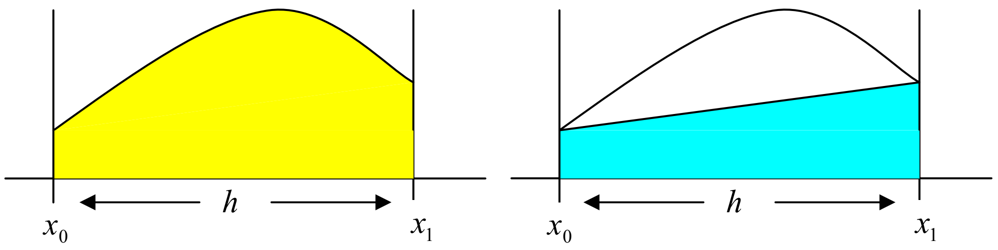


Figure 4: The trapezoidal rule

The trapezoidal rule approximates the area on the left side of figure 4 with the area on the right side. The approximation is

$$\int_{x_0}^{x_1} f(x) \quad dx \approx \frac{h}{2}\left(f(x_0) + f(x_1)\right).$$

where we have set $h = x_1 - x_0$ to be the size of the interval.

Before we can use the trapezoidal rule, we need to know how accurate it is. We can find out by using Taylor's Theorem. Indeed, from Taylor's Theorem, we know that

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(\xi_1)(x - x_0)^2.$$

Integrating, we find that

$$\int_{x_0}^{x_1} f(x)\,dx = \int_{x_0}^{x_1} \left\{ f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(\xi_1)(x - x_0)^2 \right\} dx$$

$$= h \cdot f(x_0) + f'(x_0) \cdot \frac{1}{2}(x - x_0)^2 \Big|_{x=x_0}^{x=x_1} + \frac{1}{2}\int_{x_0}^{x_1} f''(\xi_1)(x - x_0)^2 \, dx$$

where ξ_1 is some number depending on x. Because ξ_1 depends on x, we cannot directly evaluate the last integral.

To evaluate the second term, we use an approximation for $f'(x_0)$. We can find such an approximation with the aid of Taylor's Theorem. Indeed, there is a number ξ_2 so that

$$f(x_1) = f(x_0) + f'(x_0) \cdot h + \frac{1}{2} f''(\xi_2) h^2.$$

Solving for $f'(x_0)$, we see that

$$f'(x_0) = \frac{f(x_1) - f(x_0)}{h} + \frac{h}{2} f''(\xi_2).$$

Plugging this in above, we see that

$$\int_{x_0}^{x_1} f(x)\,dx = h \cdot f(x_0) + \frac{h^2}{2}\left[\frac{f(x_1) - f(x_0)}{h} + \frac{h}{2} f''(\xi_2)\right] + \frac{1}{2}\int_{x_0}^{x_1} f''(\xi_1)(x - x_0)^2 \, dx$$

$$= \frac{h}{2}\left(f(x_0) + f(x_1)\right) + E$$

where the error E is

$$E = \frac{h^3}{4} f''(\xi_2) + \frac{1}{2}\int_{x_0}^{x_1} f''(\xi_1)(x - x_0)^2 \, dx.$$

Thus

$$|E| \le \frac{h^3}{4} \max|f''| + \frac{1}{2} \max|f''| \int_{x_0}^{x_1} (x - x_0)^2 \, dx$$

$$\le \max|f''| \left[\frac{h^3}{4} + \frac{h^3}{6}\right]$$

$$\le Ch^3 \max|f''|.$$

We have proven that

$$\int_{x_0}^{x_1} f(x)\quad dx = \frac{h}{2}\left(f(x_0)+f(x_1)\right)+E$$

where

$$|E| \le Ch^3 \max|f''|$$

where $C = \frac{1}{4}+\frac{1}{6}=\frac{5}{12}$. The result is actually true for $C=\frac{1}{12}$, but the proof is more complex.

The problem with this approach is that the error gets larger as the interval gets larger. Our solution is to split the interval up into a number of pieces, and apply the simple trapezoidal rule on each piece; the result is called the composite trapezoidal rule.

The Trapezoidal Rule. To approximate the definite integral $\int_a^b f(x)\,dx$, choose a number of subintervals n; the width of one subinterval is then $h=\dfrac{b-a}{n}$. Let $x_i = a+ih$ for $0\le i\le n$ be the endpoints of the resulting subintervals. The trapezoidal rule approximation is

$$\int_a^b f(x)\quad dx = \frac{h}{2}\left[f(x_0)+2f(x_1)+2f(x_2)+\cdots+2f(x_{n-1})+f(x_n)\right]+E$$

with error E at most

$$|E| \le \frac{h^2(b-a)}{12}\max|f''|.$$

We can prove this result by applying the simple trapezoidal rule n times, once on each subinterval. On each subinterval, there is an error of at most $\frac{1}{12}h^3\max|f''|$, and there are $n=\dfrac{b-a}{h}$ such intervals. Thus the total error is no more than

$$\frac{1}{12}h^3\max|f''|\cdot\frac{b-a}{h}$$

which is our result.

Section 3: Simpson's Rule

In practice, the trapezoidal rule is rarely used. This is because there is a much more accurate method that is nearly as simple to implement, called Simpson's rule. Suppose that we want to evaluate the definite integral $\int_{x_0}^{x_2} f(x)\ dx$ where $x_1=\frac{1}{2}(x_0+x_2)$ is the midpoint of the interval $[x_0,x_2]$, and $h = x_1 - x_0 = x_2 - x_1$ is the width of each of the intervals $[x_0, x_1]$ and $[x_1, x_2]$. Apply Taylor's Theorem to $f(x)$ at x_1 with $n = 4$; then we find that

$$f(x) = f(x_1) + f'(x_1)(x - x_1) + \tfrac{1}{2} f''(x_1)(x - x_1)^2 + \tfrac{1}{6} f'''(x_1)(x - x_1)^3 + \tfrac{1}{24} f^{(4)}(\xi)(x - x_1)^4$$

for some ξ in the interval $[x_0, x_2]$ that depends on the value of x. Integrating this on the interval $[x_0, x_2]$, we find that

$$\begin{aligned}\int_{x_0}^{x_2} f(x)\ dx &= \int_{x_0}^{x_2} [f(x_1) + f'(x_1)(x - x_1) + \tfrac{1}{2} f''(x_1)(x - x_1)^2 \\ &\quad + \tfrac{1}{6} f'''(x_1)(x - x_1)^3 + \tfrac{1}{24} f^{(4)}(\xi)(x - x_1)^4\ ]dx \\ &= f(x_1)\underbrace{\int_{-h}^{h} dy}_{=2h} + f'(x_1)\underbrace{\int_{-h}^{h} y\ dy}_{=0} + f''(x_1)\underbrace{\int_{-h}^{h} \tfrac{1}{2} y^2 dy}_{=h^3/3} \qquad (7) \\ &\quad + f'''(x_1)\underbrace{\int_{-h}^{h} \tfrac{1}{6} y^3 dy}_{=0} + \int_{x_0}^{x_2} \tfrac{1}{24} f^{(4)}(\xi)(x - x_1)^4 dx \\ &= 2h\, f(x_1) + \tfrac{1}{3} h^3 f''(x_1) + \int_{x_0}^{x_2} \tfrac{1}{24} f^{(4)}(\xi)(x - x_1)^4 dx.\end{aligned}$$

where we have made the change of variables $y = x - x_1$.

Next we need to find an estimate of the value of $f''(x_1)$. This is similar to our position when we were deriving the trapezoidal rule. There, and here, our solution is to use the trapezoidal rule. However in this case, some additional work will be required. Apply Taylor's Theorem at x_1 to estimate the value of $f(x_0)$; we then apply Taylor's Theorem at x_1 to estimate the value of $f(x_2)$. We obtain

$$\begin{aligned} f(x_2) &= f(x_1) + hf'(x_1) + \frac{h^2}{2} f''(x_1) + \frac{h^3}{6} f'''(x_1) + \frac{h^4}{24} f^{(4)}(\xi_2), \\ f(x_0) &= f(x_1) - hf'(x_1) + \frac{h^2}{2} f''(x_1) - \frac{h^3}{6} f'''(x_1) + \frac{h^4}{24} f^{(4)}(\xi_0), \end{aligned}$$

for unknown values ξ_0 and ξ_2. If we add these equations and solve, we find that

$$f''(x_1) = \frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} - \frac{h^2}{24}\left[f^{(4)}(\xi_0) + f^{(4)}(\xi_2)\right].$$

Thus, if we substitute this into (7), we find that

$$\begin{aligned}\int_{x_0}^{x_2} f(x)\ dx &= 2h\, f(x_1) + \frac{1}{3} h^3 \left[\frac{f(x_0) - 2f(x_1) + f(x_2)}{h^2} - \frac{h^2}{24}\left[f^{(4)}(\xi_0) + f^{(4)}(\xi_2)\right]\right] \\ &\quad + \frac{1}{24}\int_{x_0}^{x_2} f^{(4)}(\xi)(x - x_1)^4 dx. \\ &= \frac{h}{3} \cdot \left[f(x_0) + 4f(x_1) + f(x_2)\right] + E\end{aligned}$$

where

$$E = \frac{-h^5}{72}\left[f^{(4)}(\xi_0)+f^{(4)}(\xi_2)\right] + \frac{1}{24}\int_{x_0}^{x_2} f^{(4)}(\xi)(x-x_1)^4\,dx.$$

Thus

$$\begin{aligned}|E| &\le \frac{h^5}{36}\max\left|f^{(4)}\right| + \frac{1}{24}\max\left|f^{(4)}\right|\int_{-h}^{h} y^4\,dy \\ &\le \left(\frac{1}{36}+\frac{1}{60}\right)h^5 \max\left|f^{(4)}\right|.\end{aligned}$$

In fact, with a more complex proof, one can show that $|E| \le \frac{1}{90}h^5 \max\left|f^{(4)}\right|$.

Like the trapezoidal rule, there is a composite version of Simpson's rule.

Simpson's Rule. To approximate the definite integral $\int_a^b f(x)\,dx$, choose an <u>even</u> number of subintervals n; the width of one subinterval is then $h = \frac{b-a}{n}$. Let $x_i = a + ih$ for $0 \le i \le n$ be the endpoints of the resulting subintervals. Simpson's rule approximation is

$$\begin{aligned}\int_a^b f(x)\quad dx = \frac{h}{3}[&f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots \\ &+2f(x_{n-2})+4f(x_{n-1})+f(x_n)]+E\end{aligned}$$

with error E at most

$$|E| \le \frac{h^4(b-a)}{180}\max\left|f^{(4)}\right|.$$

This is proven by applying Simpson's rule on the intervals $[x_0,x_2]$, $[x_2,x_4]$, ..., $[x_{n-2},x_n]$. The error of Simpson's rule on is no more than $\frac{1}{90}h^5\max\left|f^{(4)}\right|$, and we apply it $(b-a)/2h$ times.

Assignments

1. In section 1 we obtained a pair of estimates for $\sin 40^\circ$ where we had to estimate $\sin\xi$ and $\cos\xi$. In that example, we used the estimates $-1 \le \sin\xi \le 1$ and $-1 \le \cos\xi \le 1$. Explain why we could use the estimates $\frac{1}{2} \le \sin\xi \le \frac{\sqrt{2}}{2}$ and $\frac{\sqrt{2}}{2} \le \cos\xi \le \frac{\sqrt{3}}{2}$. How does this improve our result?

2. Use Taylor's Theorem with $n = 4$ to estimate $\sin 40^\circ$.

3. Use Taylor's Theorem with $n = 4$ to estimate $\cos 25^\circ$.

4. Write a program that takes as input the left endpoint a and the right endpoint b of an interval, together with a number of subintervals n. The program should return the Composite Trapezoidal Rule approximation to the integral $\int_a^b e^{-x^2}\, dx$ with n subintervals

a. Basic error checking on the values should be performed.

b. Use your program to estimate $\int_0^1 e^{-x^2}\, dx$ as accurately as possible.

c. Use your program to estimate $\int_0^\infty e^{-x^2}\, dx$ as accurately as possible. Explain the reasoning behind your answer.

5. Write a program that takes as input the left endpoint a and the right endpoint b of an interval, together with an even number of subintervals n. The program should return the Composite Trapezoidal Rule approximation and the Simpson's Rule approximation to the integral $\int_a^b \frac{\sin x}{x}\, dx$.

a. Basic error checking should be performed.

b. To avoid the (removable) singularity at the origin, you only need to be able to evaluate the integral for $a > 0$.

c. Use your program to evaluate $\int_0^\pi \frac{\sin x}{x}\, dx$ as accurately as possible.

d. Can you extend your program so that it can evaluate $\int_a^b \frac{\sin x}{x}\, dx$ for any real choices of a and b?

6. Simpson's Three-Eighth's rule for the approximation of the definite integral $\int_a^b f(x)\, dx$ is the following. Choose a number of subintervals n that is evenly divisible by *three*; the width of one subinterval is then $h = \frac{b-a}{n}$. Let $x_i = a + ih$ for $0 \le i \le n$ be the endpoints of the resulting subintervals. Simpson's Three Eighth's Rule for the approximation I_n is

$$\begin{aligned}\int_a^b f(x)dx = \frac{3h}{8}[&f(x_0)+3f(x_1)+3f(x_2)\\&+2f(x_3)+3f(x_4)+3f(x_5)\\&+2f(x_6)+3f(x_7)+3f(x_8)\\&+\dots\\&+2f(x_{n-6})+3f(x_{n-5})+3f(x_{n-4})\\&+2f(x_{n-3})+3f(x_{n-2})+3f(x_{n-1})\\&+f(x_n)]+E\end{aligned}$$

where $E = O(h^4)$.

a. Write a program that implements Simpson's Three Eighth's rule. Include basic error checking.

b. The error for Simpson's Three-Eighth's Rule is $O(h^4)$. What does this mean, and how does it compare to Simpson's Rule and the Trapezoidal Rule?

c. Would you use Simpson's Three-Eighths Rule in practice? Why or why not?

Classes and Software Design

Section 1: Object-oriented programming

In object-oriented programming paradigm, we group operations and data into modular units called *objects*. The operations define the object interface and describe a set of commands and actions that an object can perform and respond to. The data segment of an object is used to maintain the state of the object. The main design philosophy of object-oriented programming paradigm is to use objects to represent and model as closely as possible the way real-world entities interact. Thus, objects and object interactions are the basic elements of this model. Another salient feature of object-oriented programming is code reusability and extendibility. If an object is defined in our environment, it can be used in different applications. Also, we can extend the definition of an object to define new object types. This property is called inheritance.

Section 2: Classes

Classes are predefined types in C++ and are used to define new types. An object is instance of a class. A class has two parts: data segment and its associated operations. The operations are called *member functions* or *methods* and the variables where the data is stored are called *data members* of the class. Each object maintains its own copy of the data members. Collectively, the data members define the state of the object.

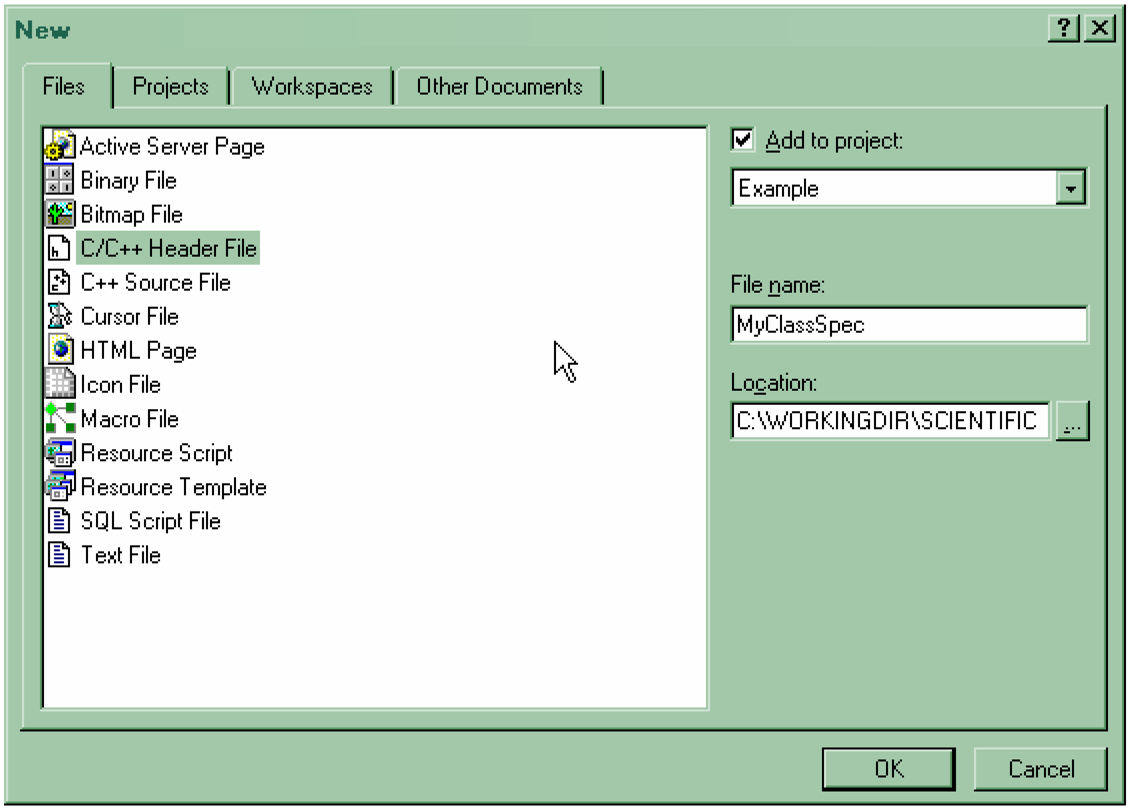
Two separate files are used for class type declaration: a specification file and an implementation file. The specification file contains the class interface which consists of the declaration of the class variables and functions. The implementation file contains the implementation of the class member functions. The class specification is stored in a header file, denoted by file extension .h, and is available to any client code that wants to use the class. The class implementation is stored in a source file, denoted by file extension .cpp. Separation of the class specification from its implementation allows a more efficient execution of the programs. Changes to the implementation file will not require modification of the client code, as long as the class interface remains the same. Moreover, this provides an added security by not requiring the client code to have access to the implementation file.

The syntax for the class specification is:

```
class  myClass
{
   public:

      constructor functions
```

```
                member functions
                class variables
                destructor functions
            private:
                data variables
                private functions
        }
```

To create a header file, from Visual C++, select File, then New. A dialog box like the one in Figure 1 will be displayed on your screen. The given types are the different file types that one can create. Select C/C++ Header File and provide a file name. The new file will be added to the selected project. The projected name is displayed in the box on the top right corner of the dialog box.



Add to project: Example

File name: MyClassSpec

Location: C:\WORKINGDIR\SCIENTIFIC

Figure 1: Creating a header file

To create a C++ source file, we can repeat the above steps and select C++ source file. The implementation file contains the definition of the member functions. The syntax for a member function is similar to regular function definitions. The only difference is that we have to qualify the name of the function by using the resolution operator "::" and the class name.

```
return_type  myclass::functionname(param list)
{
      // Do something here…
}
```

A class header file should be included in its implementation file and all the other client files that use that particular class.

What we described in this section is the manual way of declaring and creating classes. The programmer actually creates the files and manages any changes to these files. There is another, simpler, way to create classes and that is by using the MFC class wizard, which is described in section 6.

Section 3: Constructors and Destructors

A class constructor is a member function whose purpose is to initialize the private data members of a class object. The name of a constructor is always the name of the class, and there is no return type for the constructor. A class may have several constructors with different parameter lists. A constructor with no parameters is the default constructor. A constructor is implicitly invoked when a class object is declared. As a matter of style, the constructor function declarations are normally positioned at the top of the class public section. A constructor function does not have a return type and since it is implicitly called when an instance of a class is created we never directly call a constructor function.

A class destructor is a member function whose purpose is to provide a clean exit when an instance of a class ceases to exit. We do not need to provide a destructor function for classes that do not allocate memory dynamically, as the memory for automatic data objects are deallocated when the instance ceases to exist. However, if an instance of a class allocates memory dynamically for its data objects, one needs to provide a destructor function to deallocate the memory before it exits. Similar to the constructor functions, a destructor function does not have a return call and cannot be called directly by the programmer. It is called automatically when the object goes out of scope and ceases to exist. The general syntax of a destructor function declaration is

```
~class_name ( )
{
      Deallocate memory for all the dynamically allocated
      structures.
}
```

Section 4: Access Control for Variables & Functions

For each variable and function declared in a class we have to provide an access method defining who can access these data objects and functions. The possible access methods are:

- **Public:** Data and functions declared in the public section of a class are available to client programs. In other words they are visible from outside of the class.

- **Private:** Data and functions declared in the private section of the class are only accessible within the class and are not available to the client programs. Private is the default access control for the class data and function, unless specified otherwise.
- **Protected:** Data and functions declared in the protected section are only available to the derived classes (described in Section 5).
- **Friend:** Functions and class types declared as friend of a class have access to the private data and function members of the class.

In general the access method for the data members of class is private. We do not want an object's data to be modified directly by the client code. To have control over the state of the object any access to the data should be done through the member functions. Thus, it is extremely important to provide a complete set of functions that allows all possible operation on the data members. Below is a simple example of class defining a new type `student`.

```
class student     //declare the class
{
   private:
     char         name[50];
     int          ID;
     int          Num_Of_Credits;
     char         major[4];
  public:
      student(char name[], int );
      student();
      ~student();
     void  declare_major(char major[4])
     void  print_student_info();
};
```

Section 5: Inheritance and derived classes

One of the salient features of the object-oriented programming paradigm is inheritance. Though a complex subject, it can be briefly described as the ability to create new classes, referred to as derived class, from existing classes, referred to as base classes. We can define extra features in the new class by defining new data and function members and inherit the base class functions and data. The derived class has access to public and protected members of the base class. When an instance of a derived class is created, the constructor of the base class is executed first and then the constructor of the derived class. Multiple classes can be derived from the same base class. In the example below the two classes `Manager` and `Secretary` are derived classes from `Employee` class.

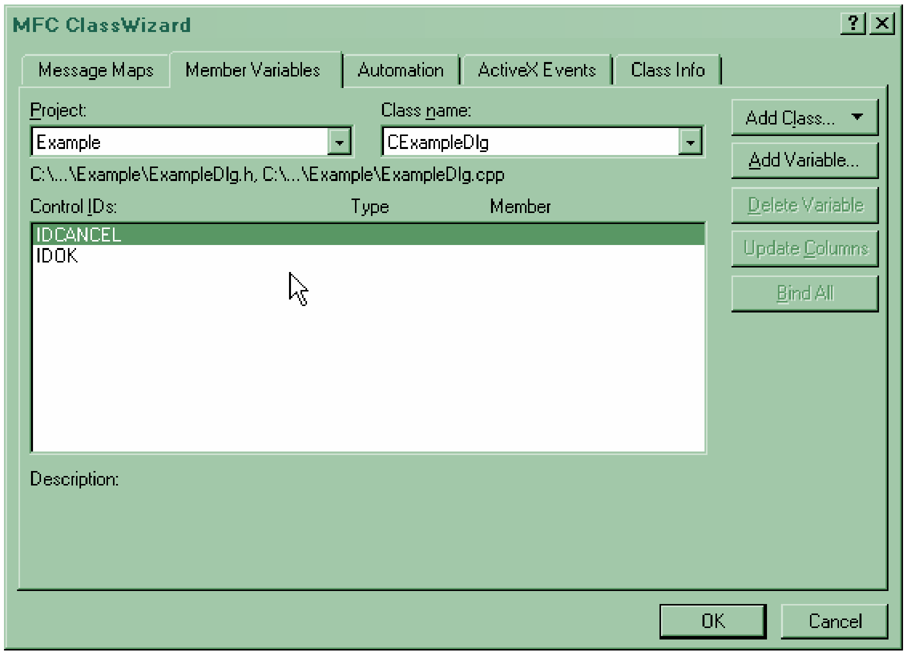
```
class Employee
{
private:
            char* name;
            int age;
            int department;
```

```
                int salary;
        protected:
                void showSalary();
        public:
                Employee( );
                void print();
        };

    //////////////////////////////////

        class Manager : public Employee
        {
            Private:
                char *      project;
                int         deptSize;
            public:
                Manager( );
                void print();
        };

    /////////////////////////////////

        class Secretary : public Employee
        {
            Private:
                char* Supervisor;
        public:
                Secretary( );
                void print();
        };
```

Section 6: Creating a Class Using the ClassWizard

In this section we describe how to use the ClassWizard to define a class type. The ClassWizard creates, maintains and manages class code. It is used to make modifications to the code, generate new derived classes and add member data and functions. It automatically adds the created files to the current project. Once classes are created using the class wizard, one should not modify the classes manually. The file with .clw extension in the project directory is created and maintained by the Cass Wizard to keep track of classes and their relationship.

Below we describe the step-by step process of creating classes using the ClassWizard. For this example create a dialog-based application, call it Example. From Visual C++, select Insert and then Dialog option. This will insert a dialog box in to your application. Another way to insert a dialog box is by selecting the ResourceView tab in the workspace pane, right click on the Dialog and select the insert option. Change the ID of dialog box to `IDD_STUDENT` and give it the caption Student. From Visual C++, select View and then ClassWizard option, or invoke the ClassWizard by directly typing CTL+W. Shown in Figure 2 is the dialog box that will be displayed on your screen.



Project: Example

Class name: CExampleDlg

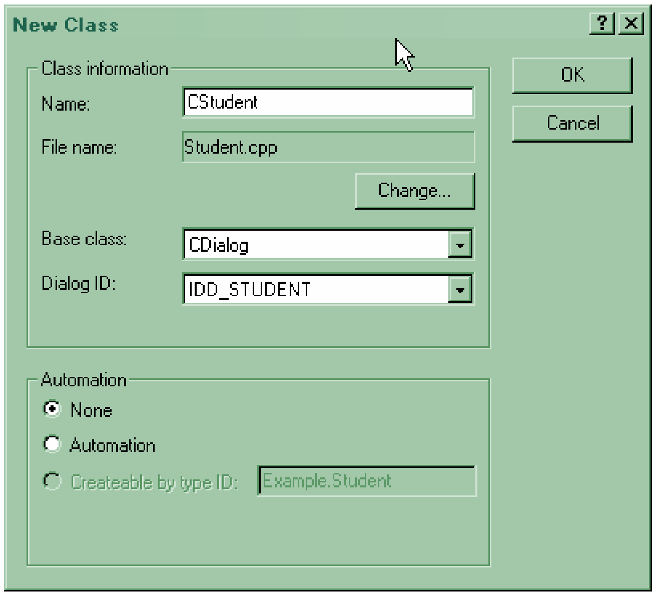
C:\...\Example\ExampleDlg.h, C:\...\Example\ExampleDlg.cpp

Control IDs:	Type	Member
IDCANCEL		
IDOK		

Description:

Figure 2: The ClassWizard

Select the Add Class button and select the first option New. The dialog box shown in Figure 3 will be displayed on your screen. Provide a name and a base class for the newly defined class type. In this case we named the class type `CStudent` and it is derived from the base class `CDialog`. The ClassWizard creates the `CStudent` class by creating the corresponding source and header files. We can add member functions and member variables by right clicking on the class name that appears in the ClassView of the Workspace pane, shown in Figure 4.



Class information

Name: CStudent

File name: Student.cpp

Base class: CDialog

Dialog ID: IDD_STUDENT

Automation

[✓ None]

Automation

Createable by type ID: Example.Student

Figure 3: The New Class dialog

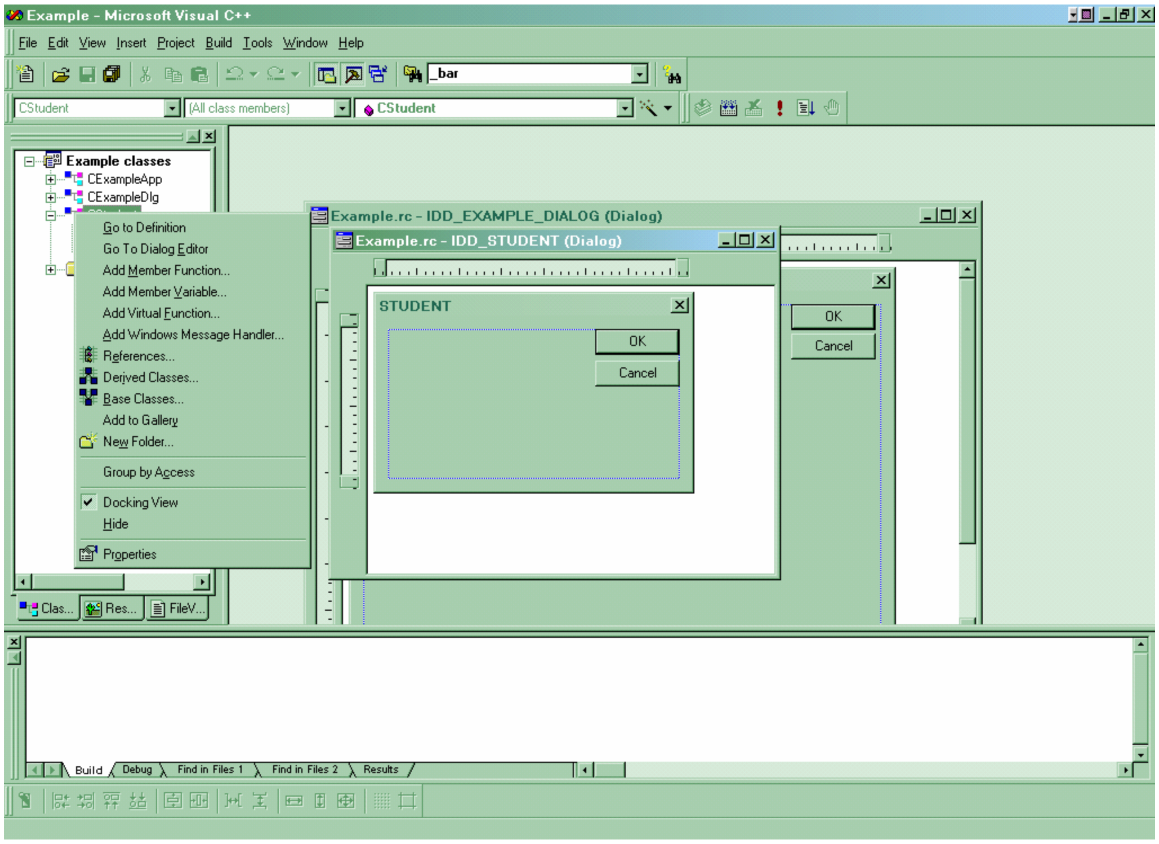
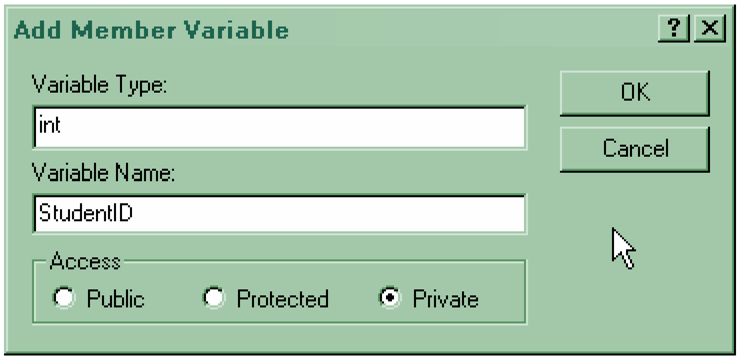


Figure 4: Right-click on a class

The following dialog box is displayed when Add Member Variable is selected. In this example, we declare `StudentID` a private variable of integer type.



Variable Type: int

Variable Name: StudentID

Access: Public Protected [✓ Private]

Figure 5: Add Member Variable dialog

Similarly, we can declare a new function by selecting add member function option.

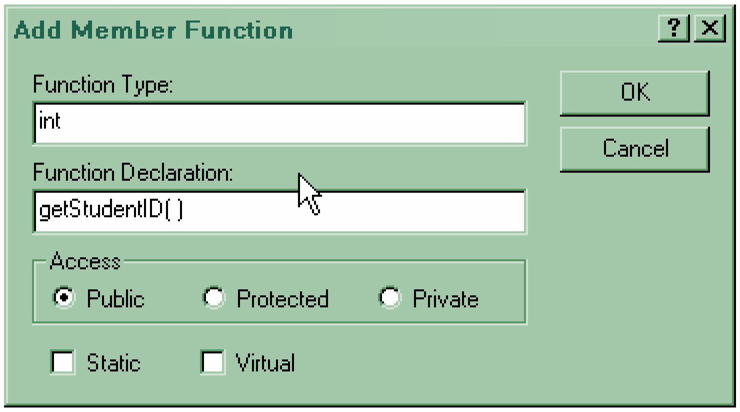


Figure 6: Add Member Function

Below lists the content of the source and header files automatically generated by the ClassWizard after the above declarations:

Header File:

```
#if !defined(AFX_STUDENT_H__EF84C76A_32B5_
      4779_A593_EF886F02D186__INCLUDED_)
#define AFX_STUDENT_H__EF84C76A_32B5_4779_
      A593_EF886F02D186__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// Student.h : header file
//

/////////////////////////////////////////////////////////////////
// CStudent dialog

class CStudent : public CDialog
{
// Construction
public:
      int getStudentID( );
      CStudent(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
      //{{AFX_DATA(CStudent)
      enum { IDD = IDD_STUDENT };
            // NOTE: the ClassWizard will add data members here
      //}}AFX_DATA


// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CStudent)
      protected:
      virtual void DoDataExchange(CDataExchange* pDX);
                        // DDX/DDV support
      //}}AFX_VIRTUAL
// Implementation
protected:

      // Generated message map functions
      //{{AFX_MSG(CStudent)
            // NOTE: the ClassWizard will add
            //    member functions here
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
private:
      int StudentID;
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
//    immediately before the previous line.

#endif // !defined(AFX_STUDENT_H__EF84C76A_32B5_
       //         4779_A593_EF886F02D186__INCLUDED_)
```

Source File:

```
// Student.cpp : implementation file
//

#include "stdafx.h"
#include "Example.h"
#include "Student.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////
// CStudent dialog


CStudent::CStudent(CWnd* pParent /*=NULL*/)
      : CDialog(CStudent::IDD, pParent)
{
      //{{AFX_DATA_INIT(CStudent)
            // NOTE: the ClassWizard will add member
                  initialization here
      //}}AFX_DATA_INIT
}


void CStudent::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(CStudent)
            // NOTE: the ClassWizard will add DDX and DDV calls
            // here
      //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CStudent, CDialog)
      //{{AFX_MSG_MAP(CStudent)
            // NOTE: the ClassWizard will add message map macros
            // here
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////
// CStudent message handlers

int CStudent::getStudentID()
{

}
```

Differential Equations and Numerical Methods

Section 1: Introduction to Differential Equations

Many physical problems can be modeled with the aid of differential equations. As an example, let $y(t)$ be the height of a ball above the ground. We know from physics that near the ground, the acceleration due to gravity is a constant $g = 9.80 \text{ m/s}$. A simple model of the motion of the ball would then be

$$y''(t) = -g$$

If we can solve this differential equation, we can then simulate the motion of the ball. We shall encounter many such differential equations in our course. Before we continue however, we would like to introduce some of the terminology of differential equations. We then discuss some numerical methods that can be used to solve differential equations.

A differential equation is an equation where the unknown is a function whose derivatives are present in the equation. Examples of differential equations include

- $y'' = -g$
- $y'' + y = 0$
- $y' + y = 0$
- $t^2 y'' + ty' + t^2 y = 0$

In general, it is difficult to solve a differential equation analytically; in fact though there are closed form solutions of the first three examples, the nonzero solutions of the fourth can not be written as any finite combination of rational functions, trigonometric functions, inverse trigonometric functions, exponential functions, or logarithmic functions.

The *order* of a differential equation is the highest number of derivatives of the unknown that appear in the equation. Three of our examples are second order and one is first order. (Which equation is first order?) We begin our study by concentrating on first order equations; we will return to higher order equations later.

Even a first order equation can be unsolvable; as an example, consider the equation

$$\sqrt{y'(t)+1} = -1.$$

This equation has no solution because the square root of any real number can never be negative. However, the reason that this problem has no solution is not that it is a differential equation, but rather because of its algebraic form. Thus, to avoid difficulties that are essentially algebraic in nature, we will assume that all of our first order equations can be solved for the derivative of the unknown. As a consequence, we shall restrict our attention to first order differential equations of the form

$$y'(t) = f\left(t, y(t)\right).$$

As examples of this type of problem, we have the following:

- $y' = y$
- $y' = y^2 + 2y + 1$
- $y' = t^2 + t - 1$
- $y' = \sqrt{y^2 + t - 1}$

Note that all of these are in the form $y'(t) = f\left(t, y(t)\right)$. Indeed

- For $y' = y$ we have $f(t, y) = y$;
- For $y' = y^2 + 2y + 1$ we have $f(t, y) = y^2 + 2y + 1$;
- For $y' = t^2 + t - 1$ we have $f(t, y) = t^2 + t - 1$;
- For $y' = \sqrt{y^2 + t - 1}$ we have $f(t, y) = \sqrt{y^2 + t - 1}$.

Recall that a function can be of the form $f(t, y)$ even if it has no explicit dependence on t or on y, as we see in three of our four examples.

Section 2: Initial Value Problems

Consider a culture of bacteria, and let $y(t)$ be the number of bacteria present at time t. We know that, given adequate living conditions, these bacteria will reproduce, and that their rate of growth is proportional to the number of bacteria already present. This yields a model of the form

$$y'(t) = k \cdot y(t)$$

for some unknown constant k; for simplicity we will assume that $k = 1$ so that our model becomes

$$y'(t) = y(t).$$

This problem can be solved by inspection- indeed what simple function is equal to its own derivative? One solution is the function $y(t) = e^t$, but this is not the only solution. The function $y(t) = 2e^t$ is a solution, as are $y(t) = -e^t$ and $y(t) = 9875e^t$. In fact, for any constant C, the function $y(t) = Ce^t$ is a solution of our problem. This is a *one-parameter* family of solutions. In general first order equations will have a one-parameter family of solutions, while second order equations will have a two-parameter family of solutions, third order equations will have a three-parameter family of solutions and so on. [It should be noted that, while this is true in general, it is <u>not</u> always the case!]

Can we use our model to determine the number of bacteria present when $t=1$? Unfortunately the answer is no; not enough information has been given. The best we can say is that the solution is $y(t)=Ce^t$ for some constant C, but we do not know the value of C. However, if we had one additional piece of information- say the number of bacteria at time $t=0$, then we could answer the question. Indeed, suppose that there were 1000 bacteria at time $t=0$. Then since the solution is $y(t)=Ce^t$, we can simply substitute $t=0$ and $y(0)=1000$ to discover that $C=1000$. Then we know that at time $t=1$ we have $1000e$ bacteria.

A first order *initial-value problem* is a first order differential equation together with the value of the solution at some particular time. Its general form is

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases}. \tag{1}$$

In general, our models will result in initial value problems.

Section 3: Euler's Method

Given a model in the form of an initial value problem like (1), we would like to use the computer to calculate the solution. The first method we shall learn to calculate approximate solutions is Euler's Method. We choose a step size h, and then consider the times $t_i = t_0 + ih$. We would like to find approximations y_i

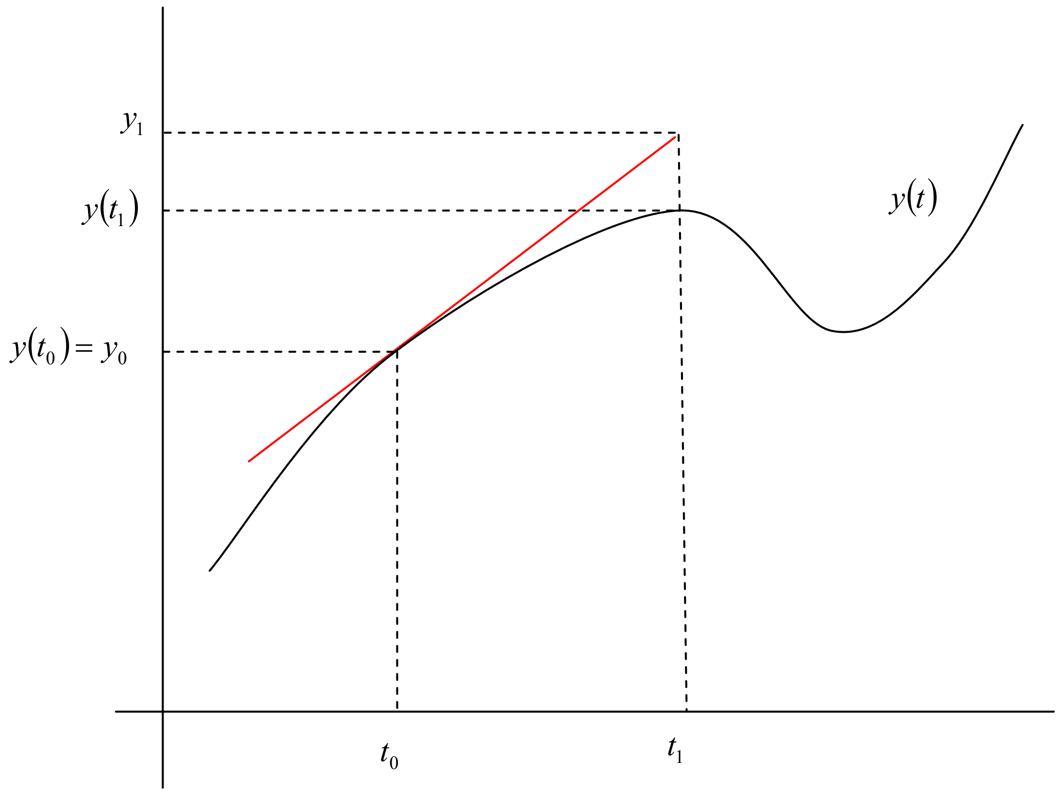


Figure 1: Euler's Method: The First Step

of our solution so that $y_i \approx y(t_i)$.

From the initial value problem (1), at time t_0, we know that the value of the solution is y_0. The differential equation then tells us the slope of the solution at t_0 is $y'(t_0) = f(t_0, y_0)$. We then proceed along the tangent line (drawn in red in figure 1) to time t_1 to obtain the value y_1. Since the tangent line approximates the function $y(t)$, the value y_1 approximates the function y at time t_1.

The formula for y_1 is simple to derive. Because the red line is the tangent line to $y(t)$ through (t_0, y_0), its slope is $y'(t_0) = f(t_0, y_0)$. On the other hand, because the red line passes through (t_0, y_0) and (t_1, y_1), its slope is

$$\frac{y_1 - y_0}{t_1 - t_0} = \frac{y_1 - y_0}{h}.$$

Thus

$$\frac{y_1 - y_0}{h} = f(t_0, y_0)$$

and hence

$$y_1 = y_0 + h \cdot f(t_0, y_0).$$

Now that we have found the approximation y_1 to $y(t_1)$, we can now try to find the approximation y_2 of $y(t_2)$ where $t_2 = t_1 + h = t_0 + 2h$. To do so, we use the same process. In particular, if the solution passes through $(t_1, y(t_1))$, then its tangent line through (t_1, y_1) has slope $y'(t_1) = f(t_1, y_1)$. We can proceed along this tangent line to time t_2 to obtain the value . As above, we see that

$$y_2 = y_1 + h \cdot f(t_1, y_1).$$

We can then proceed inductively. The values of t_i are determined by

$$t_i = t_0 + i \cdot h \tag{2}$$

while the values of our approximations are given by

$$y_i = y_{i-1} + h \cdot f(t_{i-1}, y_{i-1}) \tag{3}$$

beginning with $i = 1$.

To see how this process proceeds, consider the following initial value problem:

$$\begin{cases} y'(t) = \dfrac{2 \cdot t}{y} \\ y(0) = 1 \end{cases} \tag{4}$$

To calculate the approximation, we can choose any positive step size; we shall choose a step size of $h = 0.1$. We shall implement four steps of Euler's method and find the approximations to $y(0.1)$, $y(0.2)$, $y(0.3)$ and $y(0.4)$.

Comparing (4) with (1), we see that $t_0 = 0$, $y_0 = 1$ and $f(t, y) = \frac{2 \cdot t}{y}$. From (2), we find that $t_1 = 0.1$, $t_2 = 0.2$, $t_3 = 0.3$, and $t_4 = 0.4$. Next, we use (3) to see that

$$y_1 = y_0 + h \cdot f(t_0, y_0) = 1 + 0.1 \cdot \frac{(2) \cdot (0)}{1} = 1$$
$$y_2 = y_1 + h \cdot f(t_1, y_1) = 1 + 0.1 \cdot \frac{(2) \cdot (0.1)}{1} = 1.02$$
$$y_3 = y_2 + h \cdot f(t_2, y_2) = 1.02 + 0.1 \cdot \frac{(2) \cdot (0.2)}{1.02} = 1.05922$$
$$y_4 = y_3 + h \cdot f(t_3, y_3) = 1.05922 + 0.1 \cdot \frac{(2) \cdot (0.3)}{1.05922} = 1.11587$$

If we compare these approximations to the exact solution $y(t) = \sqrt{1 + 2t^2}$, we obtain the following table.

Approximate Solution	**Exact Solution**	**Error**
$y_1 = 1$	$y(0.1) = 1.00995$	$\lvert y_1 - y(0.1)\rvert = 0.00995$
$y_2 = 1.02$	$y(0.2) = 1.03923$	$\lvert y_2 - y(0.2)\rvert = 0.01923$
$y_3 = 1.05922$	$y(0.3) = 1.08628$	$\lvert y_3 - y(0.3)\rvert = 0.02706$
$y_4 = 1.11587$	$y(0.4) = 1.14891$	$\lvert y_4 - y(0.4)\rvert = 0.03304$

Although we have calculated the approximations, an important question is whether or not they are accurate. Before we can use Euler's method, we need to obtain some estimate of the resulting error. We can do so with the aid of Taylor's Theorem.

Indeed, consider the initial value problem (1). Apply Taylor's Theorem to the solution $y(t)$ at time t_0 to approximate $y(t_1)$. Then

$$\begin{aligned} y(t_1) &= y_0 + hy'(t_0) + \tfrac{1}{2}h^2 y''(\xi_0) \\ &= \underbrace{y_0 + hf(t_0, y_0)}_{y_1} + \underbrace{\tfrac{1}{2}h^2 y''(\xi_0)}_{E_1} \end{aligned}$$

for some unknown ξ_0, so we can write

$$y(t_1) = y_1 + E_1 \tag{4}$$

where E_1 is the (unknown error), and

$$|E_1| \le \tfrac{1}{2}h^2 \max |y''|.$$

Now apply Taylor's Theorem at t_1 to approximate $y(t_2)$. This gives us

$$y(t_2) = y(t_1) + hy'(t_1) + \tfrac{1}{2}h^2 y''(\xi_1)$$

for some unknown ξ_1. We then substitute the value for $y(t_1)$ that we obtained in (4), so that

$$\begin{aligned} y(t_2) &= (y_1 + E_1) + hf(t_1, y(t_1)) + \tfrac{1}{2}h^2 y''(\xi_1) \\ &= \underbrace{y_1 + hf(t_1, y(t_1))}_{\text{Not quite } y_2} + E_1 + \tfrac{1}{2}h^2 y''(\xi_1) \quad . \end{aligned}$$

If the second term was $hf(t_1, y_1)$ instead of $hf(t_1, y(t_1))$, then the first two terms together would be exactly y_2. However, we can add and subtract $hf(t_1, y_1)$ to see that

$$y(t_2) = \underbrace{y_1 + hf(t_1, y_1)}_{y_2} + \underbrace{E_1 + h\left[f(t_1, y(t_1)) - f(t_1, y_1)\right] + \tfrac{1}{2}h^2 y''(\xi_1)}_{E_2}.$$

Thus we can write

$$y(t_2) = y_2 + E_2$$

where

$$E_2 = E_1 + h\left[f(t_1, y(t_1)) - f(t_1, y_1)\right] + \tfrac{1}{2}h^2 y''(\xi_1).$$

How large can $|E_2|$ become? To determine this, we need to understand the behavior of the second term in out expression for E_2. We shall impose the following condition on the function $f(t, y)$. There is a constant L, called a *Lipschitz Constant* so that for every choice of t, y, and z

$$|f(t, y) - f(t, z)| \le L|y - z|.$$

In this case, we can see that

$$|f(t_1, y(t_1)) - f(t_1, y_1)| \le L|y(t_1) - y_1|$$

so that using (4) we find

$$|f(t_1, y(t_1)) - f(t_1, y_1)| \le LE_1$$

Thus

$$|E_2| \le |E_1| + hLE_1 + \tfrac{1}{2}h^2 \max|y''| \le (1 + hL)|E_1| + \tfrac{1}{2}h^2 \max|y''|.$$

We can repeat this process. Indeed for any n

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + hy'(t_n) + \tfrac{1}{2}h^2 y''(\xi_n) \\ &= (y_n + E_n) + hf(t_n, y(t_n)) + \tfrac{1}{2}h^2 y''(\xi_n) \\ &= \underbrace{y_n + hf(t_n, y_n)}_{y_{n+1}} + \underbrace{E_n + h[f(t_n, y(t_n)) - f(t_n, y_n)] + \tfrac{1}{2}h^2 y''(\xi_n)}_{E_{n+1}} \end{aligned}$$

so that

$$y(t_{n+1}) = y_{n+1} + E_{n+1}$$

where

$$E_{n+1} = E_n + h\left[f\left(t_n, y(t_n)\right) - f(t_n, y_n)\right] + \tfrac{1}{2}h^2 y''(\xi_n).$$

Thus

$$\begin{aligned}\left|E_{n+1}\right| &\le \left|E_n\right| + hL\left|y\left(t_n\right) - y_n\right| + \tfrac{1}{2}h^2 \max|y''| \\ &\le \left(1+hL\right)\left|E_n\right| + \tfrac{1}{2}h^2 \max|y''|\end{aligned}.$$

As a consequence, we shown that

$$y\left(t_n\right) = y_n + E_n$$

with the recursive relationship for the errors E_n

$$\left|E_1\right| \le \tfrac{1}{2}h^2 \max|y''|,$$

and for any $n \ge 1$

$$\left|E_{n+1}\right| \le \left(1+hL\right)\left|E_n\right| + \tfrac{1}{2}h^2 \max|y''|. \tag{5}$$

For any index i, this equation with $n = i-1$ says that

$$\left|E_i\right| \le \left(1+hL\right)\left|E_{i-1}\right| + \tfrac{1}{2}h^2 \max|y''|.$$

Thus if we apply (5) once more, now with $n = i-2$ we find that

$$\begin{aligned}\left|E_i\right| &\le \left(1+hL\right)\left[\left(1+hL\right)\left|E_{i-2}\right| + \tfrac{1}{2}h^2 \max|y''|\right] + \tfrac{1}{2}h^2 \max|y''| \\ &\le \left(1+hL\right)^2\left|E_{i-2}\right| + \left[\left(1+hL\right)+1\right]\tfrac{1}{2}h^2 \max|y''|\end{aligned}.$$

Applying (5) again, now with $n = i-3$, we find

$$\begin{aligned}\left|E_i\right| &\le \left(1+hL\right)^2\left[\left(1+hL\right)\left|E_{i-3}\right| + \tfrac{1}{2}h^2 \max|y''|\right] + \left[\left(1+hL\right)+1\right]\tfrac{1}{2}h^2 \max|y''| \\ &\le \left(1+hL\right)^3\left|E_{i-3}\right| + \left[\left(1+hL\right)^2 + \left(1+hL\right)+1\right]\tfrac{1}{2}h^2 \max|y''|\end{aligned}.$$

If we repeat this process, we find that

$$\left|E_i\right| \le \tfrac{1}{2}h^2 \max|y''|\left[1 + \left(1+hL\right) + \left(1+hL\right)^2 + \cdots + \left(1+hL\right)^{i-1}\right]$$

Thus, using the formula $1 + s + s^2 + \cdots + s^i = \dfrac{s^{i+1}-1}{s-1}$, we see that

$$\left|E_i\right| \le \frac{h\max|y''|}{2L}\left[\left(1+hL\right)^i - 1\right].$$

It is an exercise to use Taylor's Theorem to prove that $e^x \ge 1+x$ and $\left(1+x\right)^m \le e^{mx}$; thus

$$\begin{aligned}\left|E_i\right| &\le \frac{h\max|y''|}{2L}\left[e^{ihL} - 1\right] \\ &\le \frac{h\max|y''|}{2L}\left[e^{L(t_i - t_0)} - 1\right]\end{aligned}$$

where we have used the fact that $t_i - t_0 = ih$. Since $E_i = y\left(t_i\right) - y_i$, we have proven the following result.

Euler's Method. Given the initial value problem

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases},$$

assume that there is a Lipschitz constant L so that $|f(t,y) - f(t,z)| \le L|y - z|$ for every choice of t, y, and z For any step size $h > 0$, define

$$t_i = t_0 + i \cdot h$$

and

$$y_i = y_{i-1} + h \cdot f(t_{i-1}, y_{i-1}).$$

Then

$$|y(t_i) - y_i| \le \frac{h \max |y''|}{2L}\left[e^{L(t_i - t_0)} - 1\right].$$

What does this result tell us? First it tells us that the approximations produced by Euler's method do, in fact, approximate the solutions of the initial-value problem. Moreover, the smaller h becomes, the smaller the difference between y_i and $y(t_i)$. On the other hand, the farther away t_i is from t_0, the larger the difference may become.

The next thing to note is that this is how large the error *might* be, not how large the error *is*. It may be the case that the right side of the estimate

$$|y(t_i) - y_i| \le \frac{h \max |y''|}{2L}\left[e^{L(t_i - t_0)} - 1\right]$$

is large, even when the difference $|y(t_i) - y_i|$ is actually zero.

Further, the right hand side is almost never calculated in practice, in no small part because it depends on $\max|y''|$. If we know $y(t)$ well enough to calculate $\max|y''|$, we probably do not need Euler's method to construct an approximation. However, even without knowing $\max|y''|$ exactly, we can still use the result. For example, this result tells us that, when the step size is halved, the maximum error is also halved. In general, we say that if there is a constant C so that $|f(t)| \le Ct$, we say that $f(t) = O(t)$, or that f is big-*O* of t. It tells us that, if we want to double our accuracy, we will have to halve our step size- thus doubling our work.

The estimate

$$|y(t_i) - y_i| \le \frac{h \max |y''|}{2L}\left[e^{L(t_i - t_0)} - 1\right]$$

tells us that, to get the best approximation of $y(t_i)$, we should calculate approximations y_i with different values of h, and then see what happens as h tends to zero. This is how we first learned to estimate limits, by calculating the result for values of h closer and closer to zero.

There is one difficulty with this approach however. All of our analysis was predicated on the assumption that the values of y_i could be calculated exactly. When implemented on a machine however, we must think about round-off error. In fact, if there is a round-off error of no more than δ at each step, then our result must be modified to read

$$\left|y(t_n)-y_n\right| \le \left(\frac{h\max\left|y''\right|}{2L}+\frac{\delta}{hL}\right)\left[e^{L(t_n-t_0)}-1\right]+\delta e^{L(t_n-t_0)}.$$

Note that now, as h tends to zero, the error tends to infinity. Usually, this is not a concern, because the error decreases for $h<\sqrt{\frac{2\delta}{\max\left|y''\right|}}$ and only starts to increase for values of h smaller than this.

Section 4: Improving Euler's Method

There are other methods that can be used to solve first-order initial-value problems besides Euler's method. The first we shall present is called Backwards Euler, or Implicit Euler. Recall Euler's method

$$t_i = t_0 + i\cdot h$$
$$y_i = y_{i-1} + h\cdot f\left(t_{i-1}, y_{i-1}\right).$$

The slope of the line through (t_{i-1}, y_{i-1}) and (t_i, y_i) is equal to the slope provided by the differential equation at (t_{i-1}, y_{i-1}), namely $f(t_{i-1}, y_{i-1})$. In Backwards Euler, we instead require that the slope of the line through (t_{i-1}, y_{i-1}) and (t_i, y_i) is equal to the slope provided by the differential equation at (t_i, y_i), namely $f(t_i, y_i)$. This gives us the following.

Implicit Euler's Method. Given the initial value problem

$$\begin{cases} y'(t) = f\left(t, y(t)\right) \\ y(t_0) = y_0 \end{cases},$$

choose a step size h, and calculate

$$t_i = t_0 + i\cdot h$$

and

$$y_i = y_{i-1} + h\cdot f\left(t_i, y_i\right).$$

Then $\left|y(t_i)-y_i\right| = O(h)$.

The first characteristic of Implicit Euler is that it is implicit, meaning that it does not provide the value of y_i, but instead provides an equation that y_i must satisfy. We must then use some other method (like Newton's method) to actually solve the equation. This makes Implicit Euler much more difficult to implement. Further, the error for Implicit Euler is the same order as the error for Euler's method; both errors are $O(h)$.

With these facts, why introduce Implicit Euler at all? One desired characteristic of a good numerical method for solving differential equations would be that if the solution of the differential equation $y(t)$ tended to zero as $t \to \infty$, then we would expect that $y_i \to 0$ as $i \to \infty$. This is a stability condition for the method.

Consider the simple initial value problem

$$\begin{cases} y'(t) = -10y(t) \\ y(0) = A \end{cases}.$$

The solution of this problem is $y(t) = Ae^{-10t}$, which can be verified by substitution. Clearly our solution $y(t)$ tends to zero as $t \to \infty$. If we apply Euler's method to this problem, we see that

$$y_i = y_{i-1} - 10hy_{i-1} = (1 - 10h)\, y_{i-i}.$$

Thus, if we choose $h = 0.25$, we see that

$$y_i = (1 - 10 \cdot 0.25)\, y_{i-1} = -1.5y_{i-1},$$

and thus

$$|y_i| = 1.5|y_i|,$$

meaning that $|y_i| \to \infty$ as $i \to \infty$. This is clearly different than the behavior for the solution! What has occurred here is that the choice of the step size is too large for the problem; if $h < 0.2$, then we would have $y_i \to 0$ as $i \to \infty$.

Now let us compare what would occur with Implicit Euler. In this case

$$y_i = y_{i-1} - 10hy_i$$

so that

$$y_i = \frac{1}{1 + 10h}\, y_{i-1}$$

and thus $y_i \to 0$ as $i \to \infty$ for any choice of h.

The improved stability of Implicit Euler often makes up for the additional work needed to implement an implicit method. However, we shall not implement it in this course.

Euler's method requires that the slope of the line through (t_{i-1}, y_{i-1}) and (t_i, y_i) is equal to $f(t_{i-1}, y_{i-1})$ which is the slope of the solution through (t_{i-1}, y_{i-1}) while Implicit Euler uses $f(t_i, y_i)$ which is the slope of the solution through (t_i, y_i). Perhaps we can improve the accuracy of our result by using data from both points; for example by using the method

$$y_i = y_{i-1} + \frac{h}{2}\left(f(t_{i-1}, y_{i-1}) + f(t_i, y_i)\right). \tag{6}$$

Like Implicit Euler, this method is implicit, because the unknown y_i appears on both sides of the equation. One way to make this method explicit is to replace the value of y_i that appears on the right side by some approximation of y_i that we can explicitly calculate. This idea is the basis of a method called the Improved Euler method. It proceeds in two steps- first we use Euler's method to calculate an approximation to y_i; we call it $\tilde{y}_i$

$$\tilde{y}_i = y_{i-1} + hf(t_{i-1}, y_{i-1}).$$

We then use $\tilde{y}_i$ in the right side of (6) instead of y_i to obtain our final approximation

$$y_i = y_{i-1} + \frac{h}{2}\left(f(t_{i-1}, y_{i-1}) + f(t_i, \tilde{y}_i)\right).$$

The result is called the Improved Euler Method.

Improved Euler's Method. Given the initial value problem

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases},$$

choose a step size h, and calculate

$$t_i = t_0 + i \cdot h,$$

$$\tilde{y}_i = y_{i-1} + hf(t_{i-1}, y_{i-1})$$

$$y_i = y_{i-1} + \frac{h}{2}\left(f(t_{i-1}, y_{i-1}) + f(t_i, \tilde{y}_i)\right).$$

Then $\left|y(t_i) - y_i\right| = O(h^2)$.

Like Euler's Method, the Improved Euler Method is explicit and suffers from the same stability problem. The big improvement in Improved Euler however is its increased accuracy. Unlike Euler and Implicit Euler, the Improved Euler Method is accurate to $O(h^2)$. This is a dramatic improvement in accuracy. With Euler's method, if we wanted to decrease our maximum error by a factor of 100, we would need to cut the step size by 100, and do 100 times as much work. However, with Improved Euler, to decrease our maximum error by a factor of 100, we only need to cut our step size by a factor of 10, and thus we only do 10 times the work- a dramatic savings.

Section 5: Runge-Kutta Methods

Despite the increased accuracy of the Improved Euler method, it is not used in practice. Instead, a more sophisticated method, called the Runge-Kutta method of order 4 is used. It is similar in spirit to Improved Euler, but averages the slope at four different points, rather than just two.

The Runge-Kutta Method of Order 4: . Given the initial value problem

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases},$$

choose a step size h, and calculate

$$t_i = t_0 + i \cdot h .$$

To find y_i, knowing y_{i-1}, first calculate

$$\begin{aligned} k_1 &= f(t_{i-i}, y_{i-1}) \\ k_2 &= f\left(t_{i-1} + \tfrac{h}{2}, y_{i-1} + \tfrac{h}{2} k_1\right) \\ k_3 &= f\left(t_{i-1} + \tfrac{h}{2}, y_{i-1} + \tfrac{h}{2} k_2\right) \\ k_4 &= f(t_{i-1} + h, y_{i-1} + hk_3) \end{aligned}$$

and set

$$y_i = y_{i-1} + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) .$$

Then $|y(t_i) - y_i| = O(h^4)$.

This is an explicit method and the error is at most $O(h^4)$. To see the advantage of this high accuracy, note that if we were using Euler's method and wanted to decrease our maximum error by a factor of 10,000, we would need to do 10,000 times as much work; if we were using Improved Euler we would need 100 times while had we been using Runge-Kutta of order 4, just 10 times as much work would be required.

This increase in efficiency is partially offset by the fact that one step in the Runge-Kutta method of order 4 has four intermediate calculations, which is much more work than one step in Euler or in Improved Euler.

There are explicit Runge-Kutta methods of all orders, but the others are rarely used. The reason is that more and more intermediate calculations are needed to increase the order- an order 5 method needs 6 intermediate calculations, an order 6 method needs 7 intermediate calculations, and an order 7 method needs 9 intermediate calculations.

To see how Runge-Kutta works in practice, let us return to the problem (4)

$$\begin{cases} y'(t) = \dfrac{2\cdot t}{y} \\ y(0) = 1 \end{cases}$$

and use Runge-Kutta with $h = 0.1$ to approximate $y(0.1)$. We already know that $t_0 = 0$, and $t_1 = 0.1$; we also know that $f(t,y) = 2t/y$. Thus, to calculate y_1, we first find

$$\begin{aligned} k_1 &= f(t_0, y_0) = \frac{2\cdot 0}{1} = 0, \\ k_2 &= f\left(t_0 + \tfrac{h}{2}, y_0 + \tfrac{h}{2}k_1\right) = \frac{2\cdot(0+0.05)}{1+0.05\cdot 0} = 0.1, \\ k_3 &= f\left(t_0 + \tfrac{h}{2}, y_0 + \tfrac{h}{2}k_2\right) = \frac{2\cdot(0+0.05)}{1+0.05\cdot 0.1} = 0.995025, \\ k_4 &= f\left(t_0 + h, y_0 + hk_3\right) = \frac{2\cdot(0+0.1)}{1+0.1\cdot 0.995025} = 0.1980295566. \end{aligned}$$

Thus

$$y_1 = y_0 + \frac{h}{6}(k_0 + 2k_1 + 2k_2 + k_3) = 1.0099505755.$$

If we continue, we obtain the following table.

Approximate Solution	**Exact Solution**	**Error**
$y_1 = 1.00995058$	$y(0.1) = 1.00995049$	$\lvert y_1 - y(0.1)\rvert = 0.00000009$
$y_2 = 1.03923077$	$y(0.2) = 1.03923049$	$\lvert y_2 - y(0.2)\rvert = 0.00000028$
$y_3 = 1.08627858$	$y(0.3) = 1.08627805$	$\lvert y_3 - y(0.3)\rvert = 0.00000053$
$y_4 = 1.14891326$	$y(0.4) = 1.14891253$	$\lvert y_4 - y(0.4)\rvert = 0.00000073$

Note the increase in accuracy over Euler's method.

Section 6: Higher Order Equations and Systems of Equations

All of the methods was have discussed so far are for first order equations. How do we proceed with higher order equations?

Consider the equation

$$y'' + y = 0.$$

We expect that this equation has a two parameter family of solutions, and in fact, it does- the general solution is

$$y(t) = A\cos t + B\sin t$$

for any pair of constants A and B; this can be verified by substitution. To choose a particular solution, we need to determine the values of A and B; as a consequence we need two additional pieces of information.

One type of problem is called an initial-value problem. In this case, we are given the value of the function and the value of the derivative at a particular point. As an example, consider

$$\begin{cases} y''+y=0 \\ y(0)=1 \\ y'(0)=2 \end{cases}.$$

It is easy to check that the particular solution is $y(t)=\cos t+2\sin t$. No matter what values we choose for $y(0)$ and $y'(0)$, we can solve this problem.

A second type of problem is called a boundary-value problem. Here we are given values of the function at two different points. As an example, consider

$$\begin{cases} y''+y=0 \\ y(0)=0 \\ y(\pi/2)=3 \end{cases}.$$

Here, we can check that the solution is $y(t)=3\sin t$. However, we could also have the problem

$$\begin{cases} y''+y=0 \\ y(0)=0 \\ y(\pi)=0 \end{cases}.$$

If this problem can be solved, then we need to find values of A and B that satisfy the boundary conditions. Because

$$y(0) = A\cos 0 + B\sin 0 = A$$
$$y(\pi) = A\cos\pi + B\sin\pi = -A$$

we see that the solution is $y(t)=B\sin t$ for any choice of B, so this problem does not have a unique solution. On the other hand, we could also try to solve the problem

$$\begin{cases} y''+y=0 \\ y(0)=0 \\ y(\pi)=1 \end{cases}.$$

In this case, we would need $A=0$ from the condition $y(0)=0$ and $A=-1$ from the condition $y(\pi)=1$; thus this problem has no solution. As you can see, boundary-value problems need not have a unique solution; for this reason we shall not consider boundary-value problems further.

How can we solve a second-order initial-value problem? Consider the problem

$$\begin{cases} y'' + ty' + y = 0 \\ y(0) = 1 \\ y'(0) = 0 \end{cases} .$$

Our technique is to introduce a new variable, say $u(t)$, so that $u(t) = y'(t)$. If we do so, then our differential equation becomes

$$u' + tu + y = 0 .$$

Thus, if we solve, we find that we have the pair of equations

$$\begin{cases} u' = -tu - y \\ y' = u \end{cases} .$$

The key idea now, is to think of the *pair* of variables u and y as a single variable $\begin{pmatrix} u \\ y \end{pmatrix}$. Then we have the first order system

$$\begin{cases} \begin{pmatrix} u \\ y \end{pmatrix}' = \begin{pmatrix} -tu - y \\ u \end{pmatrix} \\ \begin{pmatrix} u \\ y \end{pmatrix}(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{cases} . \qquad (7)$$

We can then solve this problem using any the methods we have learned for first order equations. Let us illustrate this process by finding an approximation to $y(0.3)$ using Euler's method with a step size of $h = 0.1$.

Comparing (7) with (1), we see that $t_0 = 0$, that

$$\begin{pmatrix} u \\ y \end{pmatrix}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

and

$$f\left(t, \begin{pmatrix} u \\ y \end{pmatrix}\right) = \begin{pmatrix} -tu - y \\ u \end{pmatrix}.$$

Since $h = 0.1$, we know that $t_0 = 0.1$, $t_1 = 0.1$, $t_2 = 0.2$, and $t_3 = 0.3$. Looking at (1) once again, we see that

$$\begin{pmatrix} u \\ y \end{pmatrix}_1 = \begin{pmatrix} u \\ y \end{pmatrix}_0 + h \cdot f\left(t_0, \begin{pmatrix} u \\ y \end{pmatrix}_0\right) = \begin{pmatrix} u_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} -t_0 u_0 - y_0 \\ u_0 \end{pmatrix}$$

so that

$$\begin{pmatrix} u \\ y \end{pmatrix}_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} + 0.1 \cdot \begin{pmatrix} -0 \cdot 0 - 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -0.1 \\ 1 \end{pmatrix}.$$

Continuing, we find that

$$\begin{pmatrix} u \\ y \end{pmatrix}_2 = \begin{pmatrix} u \\ y \end{pmatrix}_1 + h \cdot f\left(t_0, \begin{pmatrix} u \\ y \end{pmatrix}_1\right) = \begin{pmatrix} u_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} -t_1 u_1 - y_1 \\ u_1 \end{pmatrix}$$

so that

$$\begin{pmatrix} u \\ y \end{pmatrix}_2 = \begin{pmatrix} -0.1 \\ 1 \end{pmatrix} + 0.1 \cdot \begin{pmatrix} -0.1 \cdot (-0.1) - 1 \\ -0.1 \end{pmatrix} = \begin{pmatrix} -0.1 \\ 1 \end{pmatrix} + \begin{pmatrix} -0.099 \\ -0.01 \end{pmatrix} = \begin{pmatrix} -0.199 \\ 0.99 \end{pmatrix} \quad .$$

Thus

$$\begin{pmatrix} u \\ y \end{pmatrix}_3 = \begin{pmatrix} -0.199 \\ 0.99 \end{pmatrix} + 0.1 \cdot \begin{pmatrix} -0.2 \cdot (-0.199) - 0.99 \\ -0.199 \end{pmatrix} = \begin{pmatrix} -0.29402 \\ 0.9701 \end{pmatrix}.$$

We can then conclude that $y(t_3) \approx y_3 = 0.9701$.

Let us also illustrate how to use approximate $y(0.1)$ using the Runge-Kutta method of order 4. In this case, k_1, k_2, k_3 and k_4 will all also be pairs.

$$k_1 = f\left(t_0, \begin{pmatrix} u \\ y \end{pmatrix}_0\right) = f\left(0, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} -0 \cdot 0 - 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

$$k_2 = f\left(t_0 + \frac{h}{2}, \begin{pmatrix} u \\ y \end{pmatrix}_0 + \frac{h}{2} k_1\right) = f\left(0.05, \begin{pmatrix} 0 \\ 1 \end{pmatrix} + 0.05 \begin{pmatrix} -1 \\ 0 \end{pmatrix}\right)$$

$$= f\left(0.05, \begin{pmatrix} -0.05 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} -0.05 \cdot (-0.05) - 1 \\ -0.05 \end{pmatrix} = \begin{pmatrix} -0.9975 \\ -0.05 \end{pmatrix}$$

$$k_3 = f\left(t_0 + \frac{h}{2}, \begin{pmatrix} u \\ y \end{pmatrix}_0 + \frac{h}{2} k_2\right) = f\left(0.05, \begin{pmatrix} 0 \\ 1 \end{pmatrix} + 0.05 \begin{pmatrix} -0.9975 \\ -0.05 \end{pmatrix}\right)$$

$$= f\left(0.05, \begin{pmatrix} -0.049875 \\ 0.9975 \end{pmatrix}\right) = \begin{pmatrix} -0.05 \cdot (-0.049875) - 0.9975 \\ -0.049875 \end{pmatrix} = \begin{pmatrix} -0.999994 \\ -0.049875 \end{pmatrix}$$

$$k_4 = f\left(t_0 + h, \begin{pmatrix} u \\ y \end{pmatrix}_0 + h k_3\right) = f\left(0.1, \begin{pmatrix} 0 \\ 1 \end{pmatrix} + 0.1 \begin{pmatrix} -0.999994 \\ -0.049875 \end{pmatrix}\right)$$

$$= f\left(0.1, \begin{pmatrix} -0.0999994 \\ 0.995013 \end{pmatrix}\right) = \begin{pmatrix} -0.1 \cdot (-0.0999994) - 0.995013 \\ -0.0999994 \end{pmatrix} = \begin{pmatrix} -0.985013 \\ -0.0999994 \end{pmatrix}$$

From the Runge-Kutta, we see that

$$\begin{pmatrix} u \\ y \end{pmatrix}_1 = \begin{pmatrix} u \\ y \end{pmatrix}_0 + \frac{h}{6}(k_0 + 2k_1 + 2k_2 + k_3)$$

$$= \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \frac{0.1}{6}\left(\begin{pmatrix} -1 \\ 0 \end{pmatrix} + 2\begin{pmatrix} -0.9975 \\ -0.05 \end{pmatrix} + 2\begin{pmatrix} -0.999994 \\ -0.049875 \end{pmatrix} + \begin{pmatrix} -0.985013 \\ -0.0999994 \end{pmatrix}\right)$$

$$= \begin{pmatrix} -0.0996667 \\ 0.980004 \end{pmatrix}.$$

We conclude that $y(t_1) \approx y_1 = 0.980004$.

Section 7: Vectors

The notion of using quantities in pair as we have done above can be made more precise. A vector $\vec{v}$ in two dimensions is a pair of real numbers, and we write $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$. We put the arrow above the letter $\vec{v}$ to remind ourselves that $\vec{v}$ is a vector. In print, vectors are often written in boldface without the arrow, giving us $\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix}$.

Vectors in two dimensions have a geometric interpretation; the line segment from the origin to the point (x, y) is $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$. Similarly, for any other point (a,b), the segment from (a,b) to $(a+x, b+y)$ is also $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$.

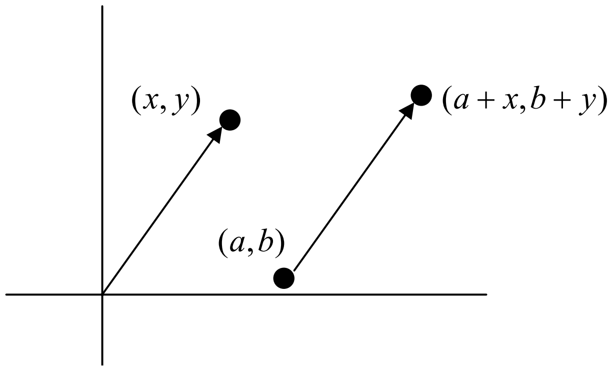


Figure 2: Vectors

We add vectors in the expected way; if $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$ and $\vec{w} = \begin{pmatrix} a \\ b \end{pmatrix}$ we define

$$\vec{v} + \vec{w} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \end{pmatrix}.$$

This has a simple geometric interpretation. If we represent $\vec{v}$ as a vector starting at the origin, and $\vec{w}$ as a vector starting at the endpoint of $\vec{v}$, then $\vec{v} + \vec{w}$ is the vector starting at the origin, and ending at the endpoint of $\vec{w}$, as seen in Figure 3.

We multiply vectors by real numbers in the expected way as well. If λ is a real number and $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$ is a vector, then the vector $\lambda\vec{v}$ is $\lambda\vec{v} = \begin{pmatrix} \lambda x \\ \lambda y \end{pmatrix}$

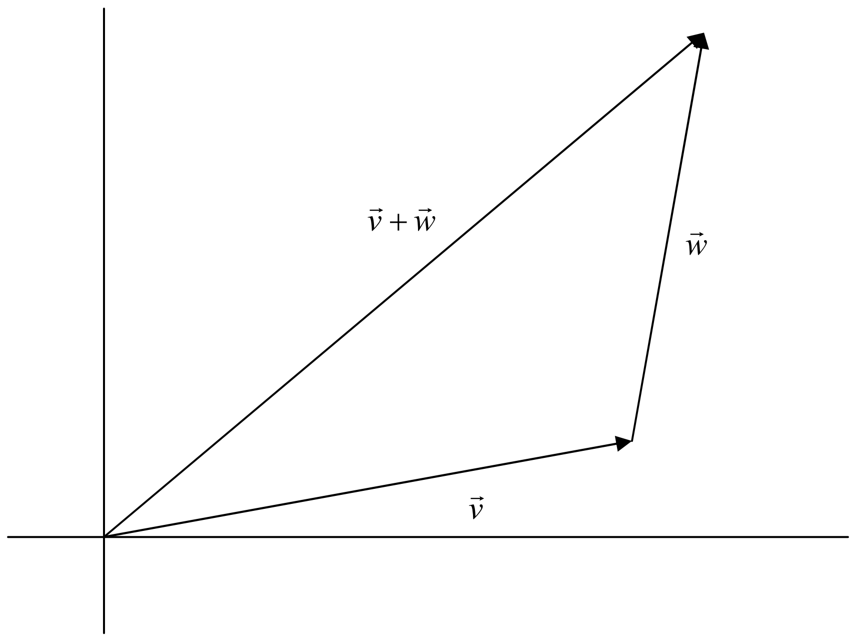


Figure 3: Adding vectors

This has a simple geometric interpretation. If we represent $\vec{v}$ as a vector starting at the origin, and $\vec{w}$ as a vector starting at the endpoint of $\vec{v}$, then $\vec{v}+\vec{w}$ is the vector starting at the origin, and ending at the endpoint of $\vec{w}$, as seen in Figure 3.

This also has a geometric interpretation. If we represent the vector $\vec{v}=\begin{pmatrix} x \\ y \end{pmatrix}$ as a

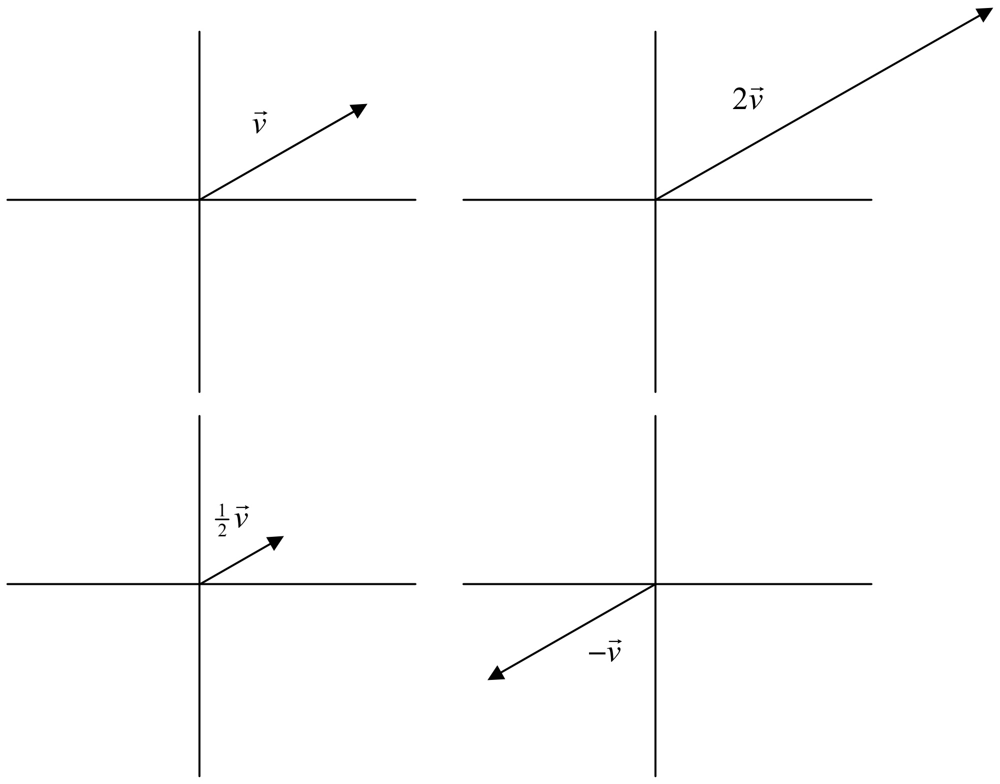


Figure 4: Multiplying vectors by real numbers

segment starting at the origin, then the vector $2\vec{v}$ can be represented as a vector starting at the origin but twice as long. Similarly, the vector $\frac{1}{2}\vec{v}$ can be represented as a vector starting at the origin but half as long, while $-\vec{v}$ can be represented by a vector starting at the origin of the same length, but pointing in the opposite direction as in Figure 4.

Finally, we define the length of the vector $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$ by

$$|\vec{v}| = \sqrt{x^2 + y^2}\,.$$

A vector of length 1 that points in the same direction as $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$ is

$$\vec{v} = \frac{1}{\sqrt{x^2 + y^2}} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \dfrac{x}{\sqrt{x^2 + y^2}} \\ \dfrac{y}{\sqrt{x^2 + y^2}} \end{pmatrix}.$$

Assignments

1. Verify the formula $1 + s + s^2 + \cdots + s^i = \dfrac{s^{i+1} - 1}{s - 1}$.

2. Use Taylor's Theorem to prove that $e^x \geq 1 + x$ for $x \geq 0$. [Hint: e^x is positive for all values of x.]

3. Prove that the error for Euler's method with round-off error decreases for $h < \sqrt{\dfrac{2\delta}{\max|y''|}}$ and increases thereafter. [Hint: Find the value of h with the minimum error.]

4. Consider the initial-value problem

$$\begin{cases} y' = yt + 1 \\ y(0) = 1 \end{cases}.$$

Use Euler's method with a step size of $h = 0.2$ and a calculator to find an approximation of $y(1)$.

5. Explain why, to solve the initial-value problem

$$\begin{cases} y'(t) = -\lambda y(t) \\ y(0) = A \end{cases}$$

for $\lambda > 0$ with Euler's method, we must impose the requirement that $|1 - h\lambda| < 1$. [Hint: What happens to $Ae^{-\lambda t}$ as $t \to \infty$? What happens to y_i as $i \to \infty$?]

6. Consider the initial-value problem

$$\begin{cases} y' = yt+1 \\ y(0)=1 \end{cases}.$$

Use Improved Euler's method with a step size of $h=0.2$ and a calculator to find an approximation of $y(1)$.

7. Consider the function $f(h)=h^2\cos h$. Is $f(h)=O(h)$? Is $f(h)=O(h^2)$? Is $f(h)=O(h^3)$? Explain.

8. Suppose we know that $f(h)=O(h^2)$. Do we know that $\left|f\left(\frac{1}{2}\right)\right| \le \frac{1}{4}$? Explain.

9. Consider the initial-value problem

$$\begin{cases} y' = yt+1 \\ y(0)=1 \end{cases}.$$

Use the fourth-order Runge-Kutta method with a step size of $h=0.2$ and a calculator to find an approximation of $y(0.2)$.

Problems 10-12 refer to the initial-value problem

$$\begin{cases} y''+y'+y-t+1=0 \\ y(0)=0 \\ y'(0)=1 \end{cases}.$$

10. Introduce an auxiliary variable, and write this problem as a first order system.

11. Use Euler's method with a step size of $h=0.2$ and a calculator to obtain an approximation to $y(1)$.

12. Use the Runge-Kutta method of order four with a step size of $h=0.2$ and a calculator to obtain an approximation to $y(0.2)$.

13. Consider the initial-value problem

$$\begin{cases} y' = t+\sqrt{y^4+1} \\ y(0)=0 \end{cases}.$$

Write a computer program that takes as input a step size, and a value of t. Your program should then calculate the Euler method approximation to $y(t)$, the Improved Euler approximation to $y(t)$, and the Fourth-Order Runge-Kutta approximation of $y(t)$.

14. Consider the initial-value problem

$$\begin{cases} y'(t)=1-t^2-y \\ y(0)=1 \end{cases}.$$

Write a computer program that takes as input a step size, and a value of t. Your program should then calculate the Euler method approximation to $y(t)$, the

Improved Euler approximation to $y(t)$, and the Fourth-Order Runge-Kutta approximation of $y(t)$. Compare your results with the exact solution $y(t) = 2e^{-t} - t^2 + 2t - 1$ for different values of the step size h. Which method is most accurate as $h \downarrow 0$? Explain your answer.

15. Consider the initial-value problem

$$\begin{cases} y''(t) + y(t) = \sin t \\ y(0) = 0 \\ y'(0) = 1 \end{cases}.$$

Write a computer program that takes as input a step size, and a value of t. Your program should then calculate the Euler method approximation to $y(t)$ and the Fourth-Order Runge-Kutta approximation of $y(t)$. Compare your result to the exact solution $y(t) = \frac{3}{2}\sin t - \frac{t}{2}\cos t$ for different values of the step size h. Which method is most accurate as $h \downarrow 0$? Explain your answer.

Project: The Baseball Problem

Section 1: Introduction

If you wanted to wanted to hit a baseball the farthest, at what angle would you hit it? If we ignore air resistance, this is a simple problem that is commonly solved in Calculus 2. However, in the real world we can not ignore air resistance.

Our goal is to construct a model for the motion of a real baseball, including the effects of air resistance. This model will be a system of ordinary differential equations. We can then use the numerical techniques we have already learned and write a computer program that will simulate the motion of our ball. This simulation can then be used to determine the optimal angle.

Section 2: Viscosity

How can we model the force of air resistance? We will construct our model using a surprisingly powerful tool called dimensional analysis.

We shall begin by trying to understand the nature of drag forces in fluids. When one layer of a fluid moves at a different velocity than another layer, the two layers exert a force on one another. To simplify the discussion, consider the case where we have a fixed layer of fluid, and that there is a patch of fluid at a distance y of area A moving above it with velocity v, as in Figure 1.

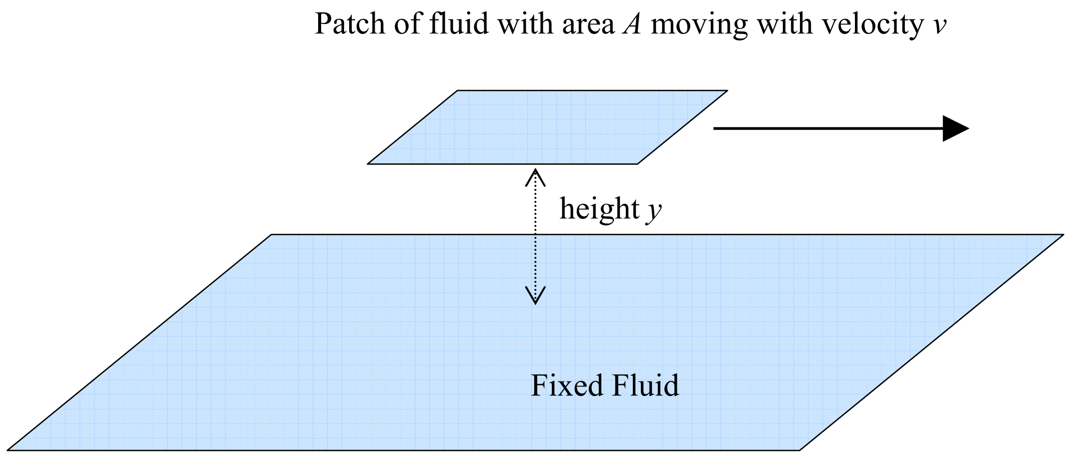


Figure 1: The physical situation

The fixed fluid exerts a drag force on the moving fluid. Clearly, the size of that drag force will depend upon the area A of the moving patch; we denote this drag force by $F(A)$. The *shear stress* of the fluid is the quantity

$$\tau = \lim_{A\to 0} \frac{F(A)}{A}.$$

If we use the MKS system of measurement, meaning that we measure using meters, kilograms, and seconds, then shear stress has units of $\mathrm{N/m^2}$, or equivalently $\mathrm{kg/m\cdot s^2}$.

The drag force depends on the velocity v of the moving patch, as well as the distance y between the moving patch and the fixed fluid. If y is large and v is small, we have a slowly moving fluid far away from the fixed fluid. In this case, we expect the drag forces to be small. On the other hand, if y is small and v is large we have a quickly moving fluid close to the fixed fluid. In this case we expect the drag force to be large. This suggests that the drag force depends on the ratio v/y. Taking the limit as $y \to 0$, we then see that the shear stress should depend on

$$\lim_{y\to 0} \frac{v}{y} = \lim_{y\to 0} \frac{v(y)-v(0)}{y} = \frac{dv}{dy},$$

where we have used the fact that when $y = 0$, we know the fluid is fixed, and thus $v = 0$.

A fluid is called *Newtonian* if the shear stress varies linearly with the derivative of the velocity. In particular, a fluid is Newtonian if there is a constant μ so that

$$\tau = \mu \frac{dv}{dy}.$$

The constant μ is called the *dynamic viscosity* of the fluid. Because $\mu = \dfrac{\tau}{dv/dy}$, we know that the MKS units for μ are $\dfrac{\mathrm{kg/m\cdot s^2}}{(\mathrm{m/s})/\mathrm{m}} = \dfrac{\mathrm{kg}}{\mathrm{m\cdot s}}$.

Many real fluids are Newtonian, including air, water, alcohol, glycerine and mercury. However, not all fluids are Newtonian; non-Newtonian fluids include toothpaste, and various paints and clays.

The dynamic viscosity μ is rarely encountered singly. More common is the ratio μ/ρ where ρ is the density of the fluid. This occurs sufficiently often that the ratio $\nu = \mu/\rho$ is called *the kinematic viscosity* of the fluid. In MKS units, because density is measured in $\mathrm{kg/m^3}$, and dynamic viscosity is measured in $\mathrm{kg/m\cdot s}$, the kinematic viscosity is measured in $\mathrm{m^2/s}$

	Dynamic Viscosity μ $(\mathrm{kg/m \cdot s})$	Kinematic Viscosity ν $(\mathrm{m^2/s})$
Water	0.0010	1.0×10^{-6}
Air	0.000018	15×10^{-6}
Alcohol	0.0018	2.2×10^{-6}
Glycerine	0.85	680×10^{-6}
Mercury	0.0016	0.12×10^{-6}

Table 1: Viscosity for common fluids at room temperature.
Source: L.D Landau & E.M. Lifshitz, *Fluid Mechanics*, Pregamon Press, 1987.

Section 3: The Drag Force on a Baseball.

With this in hand, we would like to determine the force of air resistance F_D on a moving baseball. The drag force should depend on the size of the ball, the speed of the ball, and the properties of air. This gives us the following list of quantities on which F_D might depend:

- The speed v of the baseball,
- The size of the baseball- say the diameter d,
- The density of the air ρ, and
- The viscosity of the air- say the dynamic viscosity μ.

Intuitively we expect that the mass m of the ball has an effect on the flight of the ball; this is because the acceleration a of the ball is related to the forces F acting on it by the relationship $F = ma$. Changing the mass will change the acceleration of the ball, and hence its path. However we do not expect that a change in the mass change the force that acts on the ball.

How can we use this information to determine the drag force? We know that the drag force does not care how we measure the quantities in the problem. Whether we measure speed in meters per second, or miles per hour, or even furlongs per fortnight, the drag force is the same. The number used to measure the force may be different in each case, but the actual force does not change. For this reason we look for quantities which are *dimensionless*. A physical quantity is dimensionless if it does not depend on the units of measurement that are being used. The key idea is that, dimensionless quantities, like the actual drag force, do not depend on the units in which they are measured.

To understand how dimensional analysis works, let us examine a simple example. We know that there is a relationship between the force F acting on an object, its mass m, and its acceleration a. We shall discover the form of this relationship knowing only their units of measurement. In MKS units, force is measured in N, with $1\ \mathrm{N}=1\ \mathrm{kg \cdot m/s^2}$; mass is measured in kg, and acceleration in $\mathrm{m/s}$. What dimensionless quantities can we form from the force F, the mass m, and the acceleration a? Consider the quantity

$$F^A m^B a^C$$

for unknown real numbers A, B, and C. Plugging in the units, we see that $F^A m^B a^C$ has the units

$$\left(\frac{\mathrm{kg}\cdot\mathrm{m}}{\mathrm{s}^2}\right)^A (\mathrm{kg})^B \left(\frac{\mathrm{m}}{\mathrm{s}^2}\right)^C = (\mathrm{kg})^{A+B} (\mathrm{m})^{A+C} (\mathrm{s})^{-2A-2C}.$$

If this quantity is dimensionless, then we must have

$$\begin{cases} A+B=0 \\ A+C=0 \\ -2A-2C=0 \end{cases}.$$

This has the solution $B=-A$, $C=-A$ for any choice of the number A. Thus, if we set $A=1$, we see that the quantity $Fm^{-1}a^{-1} = \frac{F}{ma}$ is dimensionless. This means that $\frac{F}{ma}$ is invariant under changes in the measuring system, and thus must be some constant. This then tells us that $F = (\text{constant})\cdot ma$. Unfortunately this is as far as dimensional analysis will take us; however measurements show us that this constant is precisely one, and hence $F = ma$.

Now we shall perform a similar analysis on the more complicated question of the drag force on a real baseball. In this case, there are five quantities in question:

Quantity		Units
Drag force exerted by the air	F_D	kg m / s2
Dynamic viscosity of the air	μ	kg / m s
Density of the air	ρ	kg / m3
Speed of the baseball	v	m / s
Diameter of the baseball	d	m

Table 2: Dimensions of the major quantities

To form a dimensionless quantity, let us examine the units of the expression $F_D^A \mu^B d^C v^D \rho^E$ where A, B, C, D, and E are unknowns. Examining Table 2, we find that this expression has units

$$(\mathrm{kg})^{A+B+E} (\mathrm{m})^{A-B+2C+D-3E} (\mathrm{s})^{-2A-B-D}.$$

Thus our expression is dimensionless if and only if

$$\begin{cases} A+B+E=0, \\ A-B+2C+D-3E=0, \\ -2A-B-D=0. \end{cases}$$

Applying the usual methods for solving linear systems of equations, we find that the solution is

$$\begin{cases} A = A, \\ B = B, \\ C = -2A - B, \\ D = -2A - B, \\ E = -A - B. \end{cases}$$

for any real numbers A and B.

Note that our solution has two arbitrary parameters, *A* and *B*. This behavior is typical when you have 3 equations in 5 variables. We can obtain two particular solutions, one with $A = 1$ and $B = 0$, and another with $A = 0$ and $B = 1$; they are

$$\begin{cases} A = 1 \\ B = 0 \\ C = -2 \\ D = -2 \\ E = -1 \end{cases} \quad \text{and} \quad \begin{cases} A = 0 \\ B = 1 \\ C = -1 \\ D = -1 \\ E = -1 \end{cases}.$$

We then discover that we can form two dimensionless quantities. The first is $\frac{F_D}{d^2 v^2 \rho}$. If we replace the square of the diameter d^2 by the cross sectional area A of the baseball, we see that $\frac{F_D}{A v^2 \rho}$ is an equivalent dimensionless quantity.

The second dimensionless quantity is $\frac{\mu}{d v \rho}$. Recall that the kinematic viscosity ν_{air} is related to the dynamic viscosity μ by $\nu_{\text{air}} = \mu / \rho$, so we can rewrite this as $\frac{\nu_{\text{air}}}{d v}$. This dimensionless ratio occurs so often in fluid dynamics that its reciprocal is called the Reynolds Number Re of the flow

$$\text{Re} = \frac{d v}{\nu_{\text{air}}} = \frac{d v \rho}{\mu}.$$

Thus, there are two dimensionless quantities for our flow, namely $\frac{F_D}{A v^2 \rho}$ and $\text{Re} = \frac{d v}{\nu_{\text{air}}} = \frac{d v \rho}{\mu}$. We conclude that there is some unknown function Φ which relates the two, so that

$$F_D = A \rho v^2 \Phi(\text{Re}).$$

Physicists define the drag coefficient $C_D = 2\Phi$ so that

$$F_D = \tfrac{1}{2} A\rho v^2 C_D(\mathrm{Re}).$$

Although dimensional analysis has shown us the form of the relationship for the drag force, the function C_D remains unknown. However, it can be found by taking careful measurement. Care must be taken, because it depends on the Reynolds number, which in turn depends on the velocity of the ball. For a smooth sphere, the drag coefficient C_D has been measured and shown to take on the following values.

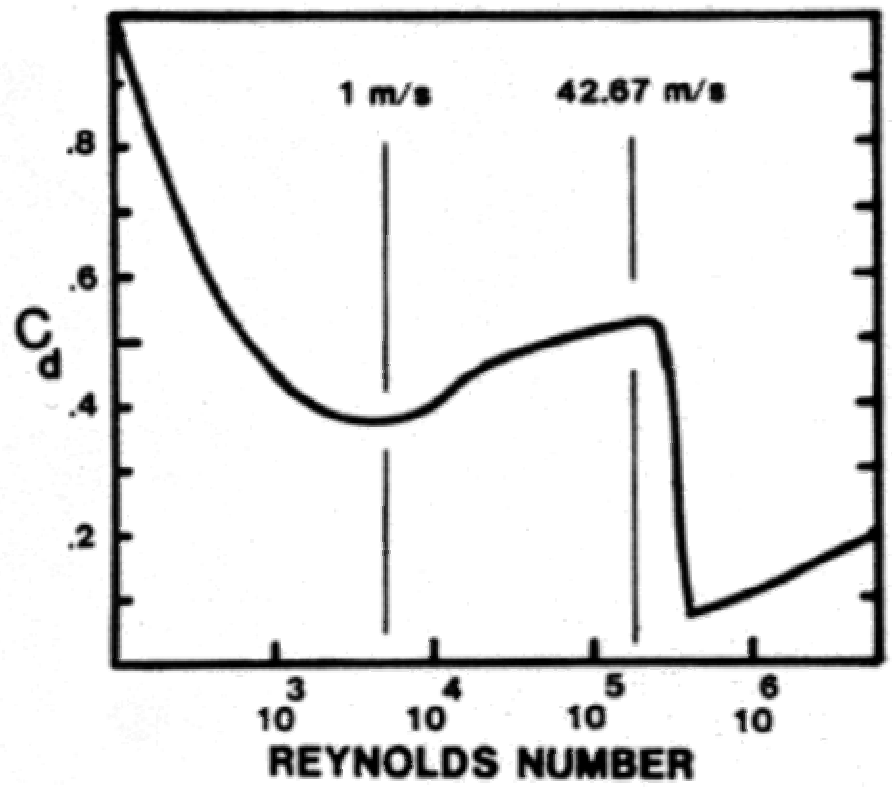


Figure 2: Drag coefficient for a smooth sphere.
Source: C. Frolich, *Aerodynamic drag crisis and its possible effect of the flight of baseballs*, Am. J. Phys. **52**(4), April 1984.

One interesting fact to note is the dramatic change in the drag coefficient from 0.5 to 0.1 that takes place near $\mathrm{Re} = 10^{5.3} \approx 2 \cdot 10^5$. This corresponds to a velocity for a smooth sphere of the same size and mass of a baseball of around 190 mi/hr.

It would be interesting to know the actual drag coefficient for a real baseball; however this is unknown. It is known that the drag coefficient C_D depends very sensitively on the degree of roughness of the ball. Below is a graph of C_D for various spheres with different degrees of roughness. Here roughness is measured as k/d where k is the height of the roughness elements, and d is the diameter of the sphere.

- Type 1: $k/d = 0.0125$;
- Type 2: $k/d = 0.0500$;
- Type 3: $k/d = 0.0150$;
- Type 4: Smooth Sphere

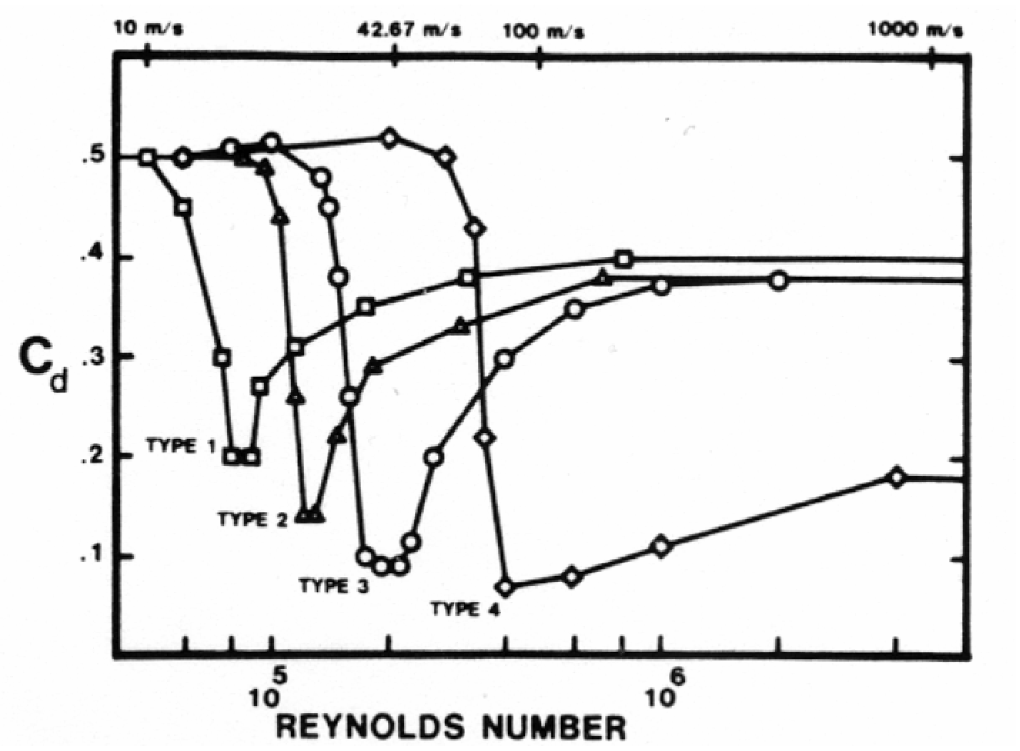


Figure 3: Drag coefficient for rough spheres.
Source: C. Frolich, *Aerodynamic drag crisis and its possible effect of the flight of baseballs*, Am. J. Phys. **52**(4), April 1984.

Note that the dramatic changes in the drag coefficient now take place much earlier; for reference note that 42.67 m/s is approximately 95 mi/hr and that 10 m/s is approximately 22 mi/hr. It is thought that the changes in the drag coefficients for a real baseball are responsible for some of the effects seen in pitched baseballs, especially for knuckleballs.

For simplicity in what follows, we shall assume $C_D = 0.5$.

Section 4: The One-Dimensional Problem

Now that we know the force of air resistance that acts on a baseball, we can begin to answer our original question- at what angle should a ball be hit so that it travels farthest? Before tacking that general problem however, we shall start by creating a model for the motion of a baseball, where we assume that the ball can only move straight up and straight down

In this case, there are only two forces that act on the baseball-

- The force of gravity $F_{\text{grav}} = mg$, and
- The drag force $F_D = \frac{1}{2} C_D A \rho v^2$.

In our model, we shall make the simplification that $C_D = 0.50$, though we have seen that, in actuality, the drag coefficient C_D varies with the Reynolds number (and hence with speed.) To finish, we need to determine the directions in which these forces act. Suppose that $y(t)$ is the height of the ball, and $v(t) = y'(t)$ is its velocity, with the distance and velocity in the upward direction being positive. We know that the force of gravity always acts downward, while the force of air resistance always acts against the motion of the ball. Thus, if the ball is moving upward, then the total force on the ball is

$$F = -mg - \tfrac{1}{2} C_D A \rho v^2.$$

On the other hand, if the ball is moving downward, then

$$F = -mg + \tfrac{1}{2} C_D A \rho v^2.$$

Because $F = ma$, where $a = y''$ is the acceleration of the baseball, we see that

$$my'' = \begin{cases} -mg - \frac{1}{2} C_D A \rho v^2 & \text{if } v = y' \geq 0 \\ -mg + \frac{1}{2} C_D A \rho v^2 & \text{if } v = y' \leq 0 \end{cases}$$

which we can simplify to obtain

$$y'' = -g - \frac{C_D A \rho v |v|}{m}.$$

For a real baseball, moving through air, we have the following values:

• $g = 9.80 \text{ m/s}^2$	the acceleration of gravity,
• $d = 0.0732 \text{ m}$	the diameter of a baseball,
• $A = \frac{1}{4}\pi d^2 = 0.00421 \text{ m}^2$	the cross sectional area of a baseball,
• $m = 0.145 \text{ kg}$	the mass of a baseball,
• $\rho = 1.29 \text{ kg/m}^3$	the density of air,
• $C_D = 0.50$	the drag coefficient.

Given our one-dimensional problem, how can we find its solution? Consider the pair of variables $\begin{pmatrix} y \\ v \end{pmatrix}$, where y is the height, and v is velocity. We know that $y' = v$; further, our model requires $y'' = -g - \frac{C_D A \rho v |v|}{m}$. Thus, we have the following system for $\begin{pmatrix} y \\ v \end{pmatrix}$

$$\begin{cases} \begin{pmatrix} y \\ v \end{pmatrix}' = \begin{pmatrix} v \\ -g - \dfrac{C_D A \rho v|v|}{m} \end{pmatrix} \\ \begin{pmatrix} y \\ v \end{pmatrix}(0) = \begin{pmatrix} y_0 \\ v_0 \end{pmatrix} \end{cases}$$

where y_0 is the initial height of the ball above the ground, and v_0 is the initial velocity of the ball. We can solve this system using the techniques we have already learned; in particular, we can use the fourth order Runge-Kutta method for systems.

Section 5: The Two-Dimensional Problem

Now let us consider the two-dimensional problem. Let x be the horizontal position of the ball, and let y be its vertical position. We can represent the position of the baseball then by the vector $\begin{pmatrix} x \\ y \end{pmatrix}$. Similarly, let the horizontal velocity of the ball be u and the vertical velocity of the ball be v; then the velocity can be represented by the vector $\begin{pmatrix} u \\ v \end{pmatrix}$.

There are two forces acting on our baseball- the force of gravity and the force of air resistance. The size of the force due to gravity is mg, and it always

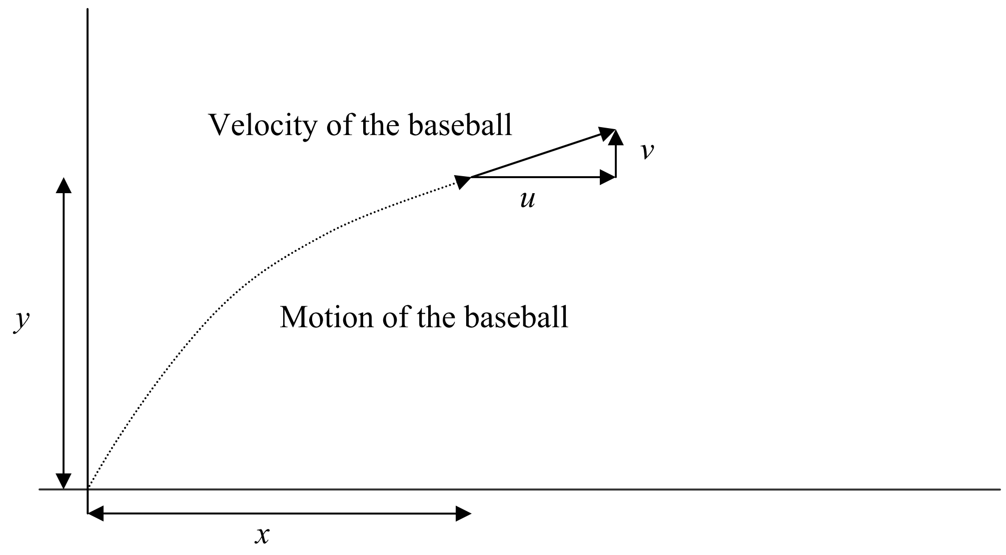


Figure 4: Variables in the Two-Dimensional Problem

acts directly downward; thus we can represent it by the vector $\begin{pmatrix} 0 \\ -mg \end{pmatrix}$.

What about air resistance? If the ball has velocity $\begin{pmatrix} u \\ v \end{pmatrix}$, then its speed is $\sqrt{u^2+v^2}$. Thus, the magnitude of the drag force is $\frac{1}{2}C_D A\rho\left(u^2+v^2\right)$. We know that the direction of the drag force is opposite the direction of motion. Consider the vector $\frac{1}{\sqrt{u^2+v^2}}\begin{pmatrix} u \\ v \end{pmatrix}$. This has length 1, and points in the direction of motion. Then a vector of size $\frac{1}{2}C_D A\rho\left(u^2+v^2\right)$ that points in the direction opposite the motion of the ball is

$$\begin{aligned} F_D &= -\tfrac{1}{2}C_D A\rho\left(u^2+v^2\right)\cdot\frac{1}{\sqrt{u^2+v^2}}\begin{pmatrix} u \\ v \end{pmatrix} \\ &= -\tfrac{1}{2}C_D A\rho\sqrt{u^2+v^2}\begin{pmatrix} u \\ v \end{pmatrix} \end{aligned}.$$

Using the relationship that $\vec{F}=m\vec{a}$, we find that our model is

$$m\begin{pmatrix} x \\ y \end{pmatrix}'' = \begin{pmatrix} 0 \\ -mg \end{pmatrix} - \tfrac{1}{2}C_D A\rho\sqrt{u^2+v^2}\begin{pmatrix} u \\ v \end{pmatrix}$$

or equivalently

$$\begin{pmatrix} x \\ y \end{pmatrix}'' = \begin{pmatrix} -\dfrac{C_D A\rho\sqrt{u^2+v^2}}{2m}u \\ -g-\dfrac{C_D A\rho\sqrt{u^2+v^2}}{2m}v \end{pmatrix}.$$

This is a second-order system of equation. To solve it, we need to convert it to a first-order system. To do so, we consider the quartet of variables $\begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix}$, and find that we have the system

$$\begin{pmatrix} x \\ y \\ u \\ y \end{pmatrix}' = \begin{pmatrix} u \\ v \\ -\dfrac{C_D A\rho\sqrt{u^2+v^2}}{2m}u \\ -g-\dfrac{C_D A\rho\sqrt{u^2+v^2}}{2m}v \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix}(0) = \begin{pmatrix} x_0 \\ y_0 \\ u_0 \\ v_0 \end{pmatrix}$$

where x_0, y_0, u_0, and v_0 are the initial positions and velocities of the ball.

Assignments

1. The period T of a pendulum with length L depends only on T, L and the acceleration of gravity g. Use dimensional analysis to find the form of the relationship.

Measurement shows us that, if $L = 9.8\text{ m}$, then $T = 4\pi\text{ s}$. Find a formula for T in terms of L and g.

2. For the one-dimensional model, write a program that takes as input

- The initial height of the baseball,
- The initial velocity of the baseball,
- An ending time, and
- A step size

and returns the fourth order Runge-Kutta method approximation to the height of the ball at the ending time. Check that your program returns reasonable values. Your program should have reasonable default values.

If the ball is thrown upward with a velocity of 90 miles/hour, how high will it be from the ground two second later?

Project

Write a C++ program that simulates the motion of a ball under air resistance.

As input, the program should take,

- The initial height of the ball,
- The speed at which it is hit,

- The angle at which it is hit, and
- A step size.

As output, the program should return

- The horizontal distance that the ball has traveled when it hits the ground.

Use your program to answer the following question- At what angle should a ball be hit to travel the farthest horizontal distance?

- A well-hit ball will travel off the bat at a speed of 110 mph.
- How accurate is your answer? How do you know that it is accurate?
- A batted ball comes off the bat with speeds between 80 mph and 130 mph. Do different initial velocities change the optimal angle? Is the change significant?

You are then to write up a technical report that answers these questions. The report should describe the model, the numerical methods used to solve the problem, your program, and your results. When answering these questions, you must address the question of how the choice of step size affects the result.

The program should be written using good object oriented programming techniques.

Graphics

Section 1: Introduction

We shall now learn how to create simple programs that use the graphical capabilities of our computers. In this chapter, we shall write a program that creates a window to hold our drawing, draws a ball in that window, and moves that ball around in a circle when a button is pressed.

Section 2: The Skeleton

We begin by creating a new dialog based program called Graphics with the AppWizard. This gives us two classes, `CGraphicsDlg` and `CGraphicsApp`, together with one window whose ID is `IDD_GRAPHICS_DIALOG`. We shall use the existing window to hold the controls for our program, and shall create a new window to hold our graphics.

To do so, we first create a new dialog window. Select the Resource Tab, Right-click on Dialog, and select "Insert Dialog". Remove the default buttons and text from your box. To identify this window, right-click and select the Properties Tab for this dialog box. Give it the name `IDD_GRAPH`.

We need to create a way for our program to access the resource we have just created. One way to do this is to create a new class that will perform the functions we would like. Use CTRL-W to start the class wizard while the new dialog box has the focus. It will tell you that `IDD_GRAPH` is a new resource, and will ask if you want to create a new class for it. Tell it to create a new class. Call this new class `CGraph`. Note that, by default it is derived from `CDialog`.

To actually use this dialog window, we need to create a variable of type `CGraph`. The simplest approach, and the one that we will take, is to create a new private variable `m_dlgGraph` of type `CGraph` in `CBallDlg`. As an alternative, note that the `CBallDlg` is instantiated in the `InitInstance()` function of the `CBallApp` class. We could do the same thing with our graphics window. Although this is cleaner from a design perspective, it does add some complexity. For example, how would the two instances of these classes pass data between them? We could pass a pointer to the first class to the constructor of the second class, so that the second class could reference the first. We shall not use this technique because of this added complexity.

If we want our program to show the dialog box we have just created, we need to tell it to do so. Because we want this new dialog box to appear when the program starts, we will add the necessary code to the OnInitDialog() method of the CBallDlg. This method is called when the dialog box is initialized for the first time. We modify it to read as follows

```
BOOL CGraphicsDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does
      // this automatically when the application's main
      // window is not a dialog
      SetIcon(m_hIcon, TRUE);            // Set big icon
      SetIcon(m_hIcon, FALSE);           // Set small icon

      // TODO: Add extra initialization here

      m_dlgGraph.Create(IDD_GRAPH);
      m_dlgGraph.ShowWindow(SW_SHOW);

      return TRUE;  // return TRUE  unless you set the
                    // focus to a control
}
```

where our addition is outlined. The first line, `m_dlgGraph.Create(IDD_GRAPH)` associates the resource `IDD_GRAPH` with the variable `m_dlgGraph`. The second tells the variable `m_dlgGraph` to execute its `ShowWindow` command, which actually causes the window to be displayed. Other options that can be passed to the ShowWindow command include

- `SW_HIDE` which hides the window,
- `SW_MINIMIZE`, which minimizes the window, and
- `SW_RESTORE` which returns the window to its original, unmaximized and unminimized position and size.

Other allowed options are described in the MSDN help system.

We can adjust the starting position of the dialog box on the screen by going to the dialog box, selecting the properties tab, and adjusting the *x*-position and *y*-position.

Section 3: Important Variable Types

When working with graphics, there are three important variable types that we need to understand. First is `CPoint`, which is just a pair of integers used to represent a point on the screen. If we wish we can pass an argument when we create our CPoint variable. Consider the code fragment:

```
CPoint Point1;
CPoint Point2(10,100);
```

In this example, `Point1` is uninitialized, while `Point2` has the value (10,100). It should be noted that, by default, Windows measures coordinates from the top right hand corner of a window, with increasing *x* coordinates moving us to the right, while increasing *y* coordinates moving us down. This differs from the usual notion of a Cartesian coordinate system, and we will need to take this into account when drawing graphs.

Our next important variable type is `CRect`, which represents a rectangular region. In addition to the default constructor, we can also use the following methods to create a `CRect`.

```
CRect rectangle1(TopLeft,BottomRight)
CRect rectangle2(left, top, right, bottom)
```

Here `TopLeft` and `BottomRight` are `CPoint` variables for the top-left and bottom-right corners of `rectangle1`. On the other hand, `left`, `top`, `right`, and `bottom` are integers describing the boundaries of `rectangle2`.

Finally, a `CDC` represents a device context. Rather than send messages about drawing objects directly to the hardware, we use a device context that handles the interaction. This means that, as programmers, we do not need to know the precise details of the hardware; moreover we can use the same set of commands to draw to a printer as to a monitor. We shall encounter `CPaintDC`, which is derived from `CDC`. Basically, this is `CDC` with added features so that we can avoid some technicalities. We will use the methods of CDC to actually draw to the screen. This is a very complex class with many different methods. We shall introduce those methods that we require as we use them.

Section 4: The Drawing Process

Because windows can be covered and uncovered as they execute, each windows program must be able to redraw itself whenever called upon to do so. Windows tells a window to redraw itself by sending a message, called `WM_PAINT`. For `CBallDlg`, this is handled by a method that the class inherits from its parent class `CDialog`. Our graphics window however, will have to override this default function.

In the class view, right-click on `CGraph`, and select "Add Windows Message Handler". Select `WM_PAINT`, and click "Add and Edit". The result is a function called `OnPaint()` which will be called whenever the `WM_PAINT` message is issued. To add some function to the `OnPaint()` method, let us begin with a simple exercise, and add the command `dc.Ellipse(0,0,20,20)` after the TODO comment, giving us the following code.

```
void CGraph::OnPaint()
{
      CPaintDC dc(this); // device context for painting

      // TODO: Add your message handler code here

      dc.Ellipse(0,0,100,100);

      // Do not call CDialog::OnPaint() for painting messages
}
```

This tells the device context to draw an Ellipse. `Ellipse` is a method of `CDC` inherited by `CPaintDC`, and can take four parameters; they are the integers (left, top, right, bottom) that represent the edges of the ellipse. If we compile and run the program, we are presented with a dialog like the one in figure 1.

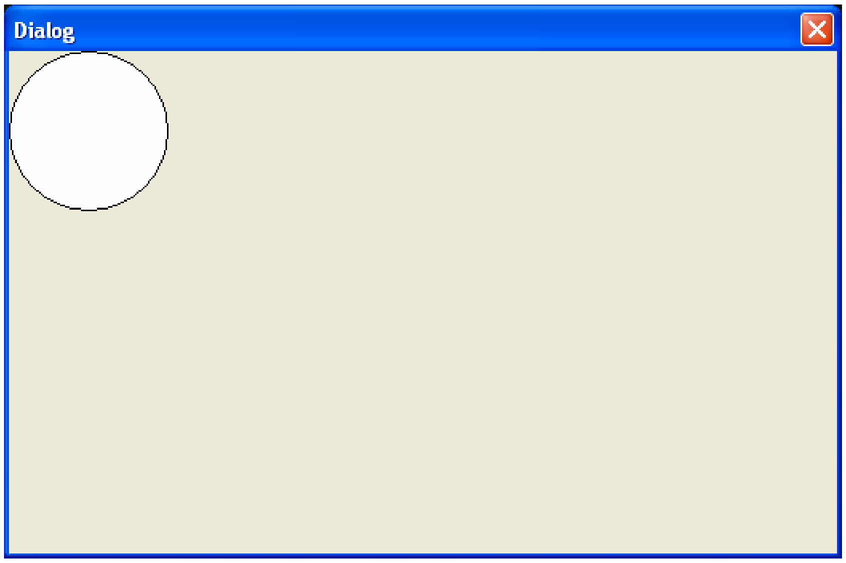


Figure 1: Result of running our simple program

Suppose that we want to change the color of the ellipse. We can do this by changing the brush. Each device context has a brush which it uses whenever called upon to draw something. However, only one brush can be used by a device context at any one time. We can define the attributes if a brush by creating an appropriate variable of type `CBrush`. Modify the `OnPaint()` method as follows.

```
void CGraph::OnPaint()
{
      CPaintDC dc(this); // device context for painting

      // TODO: Add your message handler code here

      CBrush *OldBrush;
      CBrush BlueBrush;
      BlueBrush.CreateSolidBrush(RGB(0,0,255));
      OldBrush = dc.SelectObject(&BlueBrush);
```

```
            dc.Ellipse(0,0,100,100);

            dc.SelectObject(OldBrush);

            // Do not call CDialog::OnPaint() for painting
            messages
        }
```

First we create a pointer to the `CBrush` called `OldBrush`. We will use that variable to hold the current brush. When we are finished, we will need to reset the brush back to this value. We then create a variable of type `CBrush` called `BlueBrush`; this will hold the brush we will use to draw our ellipse. Next we use the `CreateSolidBrush()` method from `CBrush` to create a solid color brush. We could also use methods like `CreateHatchBrush()` or `CreatePatternBrush()` to patterned brushes. The argument to `CreateSolidBrush`, namely `RGB(0,0,255)`, is a color directive. It specifies the amount of red, green, and blue used to create the color, where each variable lies in the range 0-255. Our brush has no red, no green, and the maximum amount of blue, so it will be a nice blue color.

Our last command in the first box loads our newly created `BlueBrush` into memory. Note that the address of `BlueBrush` is being passed as the argument. Whenever the `SelectObject` function is called, it returns a pointer to the previous value, which we store in `OldBrush`. We need to do this, because we want to make sure that we reset the brush to its original value when we are finished.

Finally, after the command to create the ellipse we have `dc.SelectObject(OldBrush)`. This returns the original brush to its place. Since we do not need to know the address of `BlueBrush` (we can get this by asking for `&BlueBrush`) we ignore the return value passed by `SelectObject`.

If we compile and run our program, we will be presented with a result like figure 1, but with a blue ball instead of a white ball.

Section 5: Animation

Now we want to animate our result. To do so, we begin by creating a variable to hold the center of the ellipse we plan to draw. Add a private variable of type `CPoint` to `CGraph`; call it `m_ptCenter`. Also add a private variable of type `int` to `CGraph` called `m_iRadius`. To initialize these variables we need to modify the constructor. Consider the following code

```
        CGraph::CGraph(CWnd* pParent /*=NULL*/)
              : CDialog(CGraph::IDD, pParent)
```

```
{
      //{{AFX_DATA_INIT(CGraph)
      // NOTE: the ClassWizard will add member
      initialization here
      //}}AFX_DATA_INIT

      m_ptCenter.x = 100;
      m_ptCenter.y = 100;
      m_iRadius = 10;
}
```

where the boxed material is new.

We now modify the OnPaint() method to use these variables to draw our ellipse rather than the hard-coded values we had used previously. Modify that method so that it now reads as follows.

```
void CGraph::OnPaint()
{
      CPaintDC dc(this); // device context for painting

      // TODO: Add your message handler code here

      CBrush *OldBrush;
      CBrush BlueBrush;
      BlueBrush.CreateSolidBrush(RGB(0,0,255));
      OldBrush = dc.SelectObject(&BlueBrush);

      CRect ball(m_ptCenter,m_ptCenter);
      ball.InflateRect(m_iRadius,m_iRadius);
      dc.Ellipse(ball);

      dc.SelectObject(OldBrush);

      // Do not call CDialog::OnPaint() for painting
      messages
}
```

where, as usual, the boxed material is new.

The new code begins by creating a rectangle `ball` whose top left corner and bottom right corner are the point m_ptCenter. This gives a 1-pixel rectangle. We then use the `InflateRect` command to increase its size while keeping it centered on the original 1-pixel rectangle. This command takes two arguments, which specify the amount the rectangle is to be increased in size, horizontally first, then vertically. Since we want our rectangle to remain a square, we give the same values to both parameters. Finally we use the `Ellipse` command to create and draw the ball. Together this draws a ball centered at `m_ptCenter`.

To move the ball around the screen, we need to do two things- change the values of the parameters `m_ptCenter` and m_iRadius, and tell Windows that we want to redraw the screen. Create a public method in `CGraph` called `SetCenter`.

It should take two integers as input and have no output. We want this function to use the two integers to determine the center of our ball, then tell windows that we want to redraw the screen. We can do this with the following code.

```
void CGraph::SetCenter(int x, int y)
{

        m_ptCenter.x = x;
        m_ptCenter.y = y;
        Invalidate();
        OnPaint();

}
```

The first two commands set the position of the ball. The third tells Windows that, the next time the window is drawn, the entire window should be drawn. The last tells windows to actually draw the window.

To see this in action, let us return to the original dialog box `IDD_GRAPHICS_DIALOG`. Remove the TODO text and the cancel button, and change the caption for the OK button to read Exit Program. Then add a new button called Animate whose ID is `ID_BUTTON_ANIMATE`. At this point we should have a main dialog box like the one in figure 2.

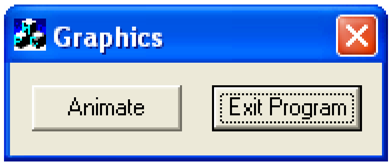


Figure 2: The main dialog box

Create the method `OnButtonAnimate()` to be executed when the Animate button is pressed. Modify that function so that it reads as follows.

```
void CGraphicsDlg::OnButtonAnimate()
{
        // TODO: Add your control notification handler code
        here

        for(int j=1000; j>=0; j--)
                m_dlgGraph.SetCenter(j/10,j/10);
        for(j=0; j<=1000; j++)
                m_dlgGraph.SetCenter(j/10,j/10);

}
```

Compile and run this code. Every time the Animate button is pressed, the ball will move to the top left corner of the window, and then back to its starting point.

Section 6: Getting Information about the Window

At this point, we can use these techniques to move a ball around the screen. However, though we have to specify the screen coordinates for our drawing, we do not know basic information about the window in which we are drawing. For example, we know neither how tall nor how wide our window is. We would like to record this information in a variable of type `CRect`. Add a private variable of type `CRect` called `m_rectView` to `CGraph`.

We would like to store information about our window in `m_rectView` as soon as the dialog is initialized. We could try to do this in the constructor. However, the constructor function is called before the window is created. (Remember how we declared the variable first, and then showed the window later?) Instead we need to do this in `OnInitDialog()` .

Examining the `CGraph` class, we note that there is no `OnInitDialog()` method. Actually, this is untrue because it inherits this method from its base class (`CDialog`). We can override this by adding a handler to the Windows Message `WM_INITDIALOG`. This handler is OnInitDialog() and we can modify it to read as follows

```
BOOL CGraph::OnInitDialog()
{
      CDialog::OnInitDialog();

      // TODO: Add extra initialization here

      GetClientRect(m_rectView);

      return TRUE;  // return TRUE unless you set the focus
                       to a control
      // EXCEPTION: OCX Property Pages should return FALSE
}
```

The new code, `GetCLientRect(m_rectView)`, stores the information about the dialog box in the variable `m_rectView`. Thus, to get information about our dialog box, we can simply access `m_rectView`. For instance, we can use any of the following

- `m_rectView.left`
- `m_rectView.right`
- `m_rectView.bottom`
- `m_rectView.top`
- `m_rectView.Width()`
- `m_rectView.Height()`
- `m_rectView.CenterPoint()`

The first six of these are integers; the last returns a `CPoint`.

Section 7: Graphs

When we want to draw a graph to the screen, we are usually know the coordinates of the points in some coordinate system, and would like the computer to produce a faithful representation on the screen. Thus, we need a method that takes as input the (x, y) coordinates of a point, and in turn draws on the screen at the corresponding point. For our example, we want the box $-1 \le x, y \le 1$ to fit on the visible screen as well as possible.

To do so, first we need to determine which is larger- the window's height or its width, and then to keep track of that result. In `CGraph`, declare a private variable of type integer called `m_iScale`. This will hold the distance from the center of our window to the nearest boundary. We also need to determine the center of the dialog; we add a variable of type `CPoint`, called `m_ptWindowCenter`. To initialize these values, modify `OnInitDialog()` as follows.

```
BOOL CGraph::OnInitDialog()
{
      CDialog::OnInitDialog();

      // TODO: Add extra initialization here

      GetClientRect(m_rectView);
      m_iScale = min(m_rectView.Width(),
                     m_rectView.Height())/2;
      m_ptWindowCenter = m_rectView.CenterPoint();

      return TRUE;  // return TRUE unless you set the focus
                       to a control
                    // EXCEPTION: OCX Property Pages should
                       return FALSE
}
```

Note the need to divide by 2 in the calculation for m_iScale. (Why?)

Now suppose that we are given a pair of numbers x and y. We need to find the corresponding screen coordinates (xsc, ysc) for that point. If $x = 0$, then

$$xsc = \text{m\_ptWindowCenter.x}\,.$$

On the other hand, if $x = 1$, then

$$xsc = \text{m\_ptWindowCenter.x} + \text{m\_iScale}\,.$$

We can then find the line through the points (0,m_ptWindowCenter.x) and (1,m_ptWindowCenter.x+m_iScale). It has equation

$$xsc - xsc_0 = \frac{xsc_1 - xsc_0}{x_1 - x_0}(x - x_0)\,,$$

which, upon substitution, can be simplified to

$$xsc = \text{m\_ptWindowCenter.x} + \text{m\_iScale} \cdot x\,.$$

Similarly, if $y = 0$, then

$$ysc = \text{m\_ptWindowCenter.y}\,.$$

If $y = 1$, then

$$ysc = \text{m\_ptWindowCenter.y} - \text{m\_iScale}\,.$$

Note the negative sign! We need the negative sign to account for the fact that, in mathematics the positive *y*-direction points up, while in Windows, the positive *y*-direction points downward. We can then find the line through the points (0,m_ptWindowCenter.y) and (1,m_ptWindowCenter.y-m_iScale). It has the equation

$$ysc - ysc_0 = \frac{ysc_1 - ysc_0}{y_1 - y_0}\left(y - y_0\right),$$

which, after substitution, can be simplified to

$$ysc = \text{m\_ptWindowCenter.y} - \text{m\_iScale} \cdot y\,.$$

With this background, we can now create a method that takes as input the (x, y) coordinates of a point, then determines the corresponding screen coordinates for that point, and sets the center of our ellipse at that point. To do so, create a public method `Draw` in `CGraph` with the following code:

```
void CGraph::Draw(double x, double y)
{

    m_ptCenter.x = m_ptWindowCenter.x
                      +(int)((double)(scale)*x);
    m_ptCenter.y = m_ptWindowCenter.y
                      -(int)((double)(scale)*y);

    Invalidate();
    OnPaint();
}
```

Note how we were very careful to explicitly cast our variable types. It is easy to make a very hard to find mistake when you rely on the compiler to cast your variables for you; sometimes it makes decisions that you do not expect.

Section 8: Circular Motion

Our original goal for this project was to create a program that moves a ball in a circle when a button is pressed. We shall replace the code for our Animate button with code that moves the ball in a circle.

We begin by initializing the position of the ball. At start, we shall draw the ball at the position $x = 1$, $y = 0$. We can do so by calling the Draw function for m_dlgGraph in the OnInitDialog() method for CGraphicsDlg. Thus, it now reads

```
BOOL CGraphicsDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does
            this automatically
      //  when the application's main window is not a
            dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon

      // TODO: Add extra initialization here

      m_dlgGraph.Create(IDD_GRAPH);
      m_dlgGraph.ShowWindow(SW_SHOW);
      m_dlgGraph.Draw(1,0);

      return TRUE;  // return TRUE  unless you set the
                        focus to a control
}
```

Next, we modify the code for the Animate button so that it draws our ball in a circle of radius 1, starting at $(1,0)$ and moving counterclockwise. We can do this by setting $x = \cos t$ and $y = \sin t$ for $0 \le t \le 2\pi$. To implement this in the code, we first make sure that we load the `math.h` header file, and then use the following code.

```
void CGraphicsDlg::OnButtonAnimate()
{
      // TODO: Add your control notification handler code
      here

//    for(int j=1000; j>=0; j--)
//          m_dlgGraph.SetCenter(j/10,j/10);
//    for(j=0; j<=1000; j++)
//          m_dlgGraph.SetCenter(j/10,j/10);

      const double pi = 4.0*atan(1.0);
      const int n = 10000;
      double x,y;
      for(int i=0; i<n; i++)
      {
            x = cos( 2.0 * (double)(i)/(double)(n) * pi);
            y = sin( 2.0 * (double)(i)/(double)(n) * pi);
            m_dlgGraph.Draw(x,y);
      }

}
```

One thing to note about this new code, is that it does not depend on the precise size of the drawing window. In particular, it only passes the (x, y) coordinates of the point where the ball is to be drawn. The process of converting that information into screen coordinates is handled by the `CGraph` class. This illustrates one of the great advantages of object-oriented programming- the separation of tasks by class. The class that decided where the ball is to be drawn does not need to know any information about the window where it is being drawn, as that entire process is handled internally by `CGraph`.

Section 9: Lines

Suppose we wanted to also draw the line from the center of the window to the moving ball. How can we do this? The shape, color, and style of lines and curves are governed by the currently selected `CPen`. The currently selected `CPen` also determines the style and color of the boundary of solid regions. You may have noticed that the balls that we have drawn all have had a thin black boundary around them; this is determined by the currently selected `CPen`.

To draw a line, we need to create a `CPen`. A `CPen` is the equivalent of `CBrush`, however where `CBrush` is used for solid regions, `CPen` is used for lines and curves. To use a `CPen`, consider the following code fragment.

```
CPen Pen;
Pen.Create(PS_SOLID,1,RGB(0,0,255));
```

The first of these commands created a `CPen` called `Pen`; the second initializes it. The first parameter tells the type of curve to be drawn. Some of the options here include

- `PS_SOLID` Creates a solid pen.
- `PS_DASH` Creates a dashed pen. Valid only when the pen width is 1 or less, in device units.
- `PS_DOT` Creates a dotted pen. Valid only when the pen width is 1 or less, in device units.
- `PS_DASHDOT` Creates a pen with alternating dashes and dots. Valid only when the pen width is 1 or less, in device units.
- `PS_DASHDOTDOT` Creates a pen with alternating dashes and double dots. Valid only when the pen width is 1 or less, in device units.

The second number is an integer which specifies the width of the Pen. The last is a color specification.

Like the `CBrush`, our `CDC` class can only hold one `CPen` at a time. Further, we must be sure to hold the pointer to the existing `CPen` so that we can re-select that `CPen` when our code completes. Here is an example of how we could use the `CPen` to select a `Pen` for drawing.

```
void CGraph::OnPaint()
{
      CPaintDC dc(this); // device context for painting
      // TODO: Add your message handler code here
```

```
          CPen *OldPen;
          CPen Pen;
          Pen.CreatePen(PS_SOLID,1,RGB(255,0,0));
          lOldPen = dc.SelectObject(&Pen);

          // Do our Drawing

          dc.SelectObject(OldPen);
    }
```

Note that the `SelectObject` method of `CDC` takes the pointer to the `Pen` as the parameter, not the `Pen` itself.

There are two commands we must learn to draw lines. The first is the command `MoveTo`; the second is the command `LineTo`. These are both methods in the `CDC` class, and are inherited by the `CPaintDC` class of our `OnPaint()` method. To see these in action, consider the following code fragment.

```
    void CGraph::OnPaint()
    {
          CPaintDC dc(this); // device context for painting
          // TODO: Add your message handler code here

          CPen *lOldPen;
          CPen Pen;
          Pen.CreatePen(PS_SOLID,1,RGB(255,0,0));
          lOldPen = dc.SelectObject(&Pen);

          dc.MoveTo(0,100);
          dc.LineTo(100,100);

          dc.SelectObject(lOldPen);
    }
```

This fragment begins by loading a 1 unit thickness red pen. The `MoveTo(0,100)` command moves the drawing position to the point with coordinates (0,100), while the `LineTo(100,100)` draws a line using the current CPen from there to the point (100,100). The `MoveTo` and `LineTo` command both will also take a CPoint as an argument instead of a pair of integers.

There are other commands that can be used to draw lines and curves, including `Arc`, and `PolyLine`. For a complete listing of the available commands, examine MSDN.

Section 10: InvalidateRect()

Thus far, we have used the `Invalidate()` command to indicate to Windows that we want the entire viewing window redrawn the next time an `WM_PAINT` message is issued. However, this is inefficient if only a portion of the screen actually needs to be redrawn. We can exert a greater degree of control using the `InvalidateRect` command.

The `InvalidateRect` command takes a `CRect` as a parameter. It tells Windows that the next time a `WM_PAINT` message is issued, to redraw the portion of the screen indicated by the `CRect`. Multiple `InvalidateRect` commands can be issued before a `WM_PAINT` message; Windows will then redraw the portion of the screen corresponding to each rectangle.

Consider the following code fragment.

```
void CGraph::OnPaint()
{
      CPaintDC dc(this); // device context for painting
      // TODO: Add your message handler code here

      CRect ball(m_ptCenter,m_ptCenter);
      ball.InflateRect(m_iRadius,m_iRadius);
      dc.Ellipse(ball);
}

void CGraph::Draw(int x, int y)
{
      CRect ball1(m_ptCenter,m_ptCenter);
      ball1.InflateRect(m_iRadius,m_iRadius);

      m_ptCenter.x = x;
      m_ptCenter.y = y;

      CRect ball2(m_ptCenter,m_ptCenter);
      ball2.InflateRect(m_iRadius,m_iRadius);

      InvalidateRect(ball1);
      InvalidateRect(ball2);

      OnPaint();
}
```

We see that this `OnPaint` method draws an ellipse at the point `m_ptCenter` of radius `m_iRadius`. Our `Draw` method is used to update the position of the ball so that we can animate its motion. The `CRect ball1` is the existing position of the ellipse, while the `CRect ball2` is the position of the ellipse when it is redrawn. As no other portion of the screen needs to be modified, we issue the two `InvalidateRect` command indicated so that only the portion of the screen that is actually being modified will be redrawn.

Assignments

1. What do we mean when we say that `CGraph` is derived from `CDialog`?
2. Experiment with different sizes of ellipses and different colors.

3. What happens if the `Invalidate()` command is omitted in our `SetCenter` method? Explain.

4. What happens if the `OnPaint()` command is omitted in our `SetCenter` method?

6. Add a button to the program that moves the ball clockwise.

7. Modify the code so that it draws a line from the center of the screen to the ball.

8. What would happen if in the code fragment for `InvalidateRect`, we removed the `InvalidateRect(ball1)` command? What would happen if we removed the `InvalidateRect(ball2)` command?

Project: The Three Body Problem

Section 1: Introduction

In this project, we will model the behavior of three bodies- say planets or stars, that move solely due to their mutual gravitational attraction. For simplicity, we will assume that the bodies are constrained to lie in a single plane.

The problem with just two bodies has been solved analytically; however, there is no analytic solution to the general three-body problem.

In the project, the student will create a computer program that will simulate the motion of three bodies under gravity, then study the resulting behavior.

Section 2: The Model

The key to our model is Newton's Law of Gravitation. It says that the gravitational attraction between two bodies of masses m_1 and m_2 located a distance r apart is

$$F = G\frac{m_1 m_2}{r^2}$$

where $G = 6.67\times 10^{-11}\ \mathrm{N}\cdot\mathrm{m}^2/\mathrm{kg}^2$ is the universal constant of gravitation.

To construct our model, let us suppose that we have three planets of masses m_1, m_2 and m_3 respectively. Since we are assuming that these planets move in a plane, we can specify their positions as $\vec{r}_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$, $\vec{r}_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$, and $\vec{r}_3 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}$.

We begin by finding the gravitational force that planet 2 exerts on planet 1. The distance between planet 1 and planet 2 is

$$\left|\vec{r}_1 - \vec{r}_2\right| = \left|\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}\right| = \sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2}$$

so that the size of the force acting on planet 1 caused by planet 2 is

$$F_{12} = G\frac{m_1 m_2}{\left|\vec{r}_1 - \vec{r}_2\right|^2} = G\frac{m_1 m_2}{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2}.$$

This force acts to pull planet 1 toward planet 2. A vector from planet 1 to planet 2 is $\vec{r}_1 - \vec{r}_2$, so that a vector of length 1 from planet 1 towards planet 2 is

$$\frac{\vec{r}_1 - \vec{r}_2}{\left|\vec{r}_1 - \vec{r}_2\right|}.$$

This then tells us that the force exerted by planet 2 on planet 1 is the vector

$$\vec{F}_{12} = G\frac{m_1 m_2}{\left|\vec{r}_1 - \vec{r}_2\right|^2}\frac{\vec{r}_1 - \vec{r}_2}{\left|\vec{r}_1 - \vec{r}_2\right|} = G\frac{m_1 m_2}{\left[\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2\right]^{3/2}}\begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \end{pmatrix}.$$

Similarly, the force exerted by planet 3 on planet 1 is the vector

$$\vec{F}_{13} = G\frac{m_1 m_3}{\left|\vec{r}_1 - \vec{r}_3\right|^2}\frac{\vec{r}_1 - \vec{r}_3}{\left|\vec{r}_1 - \vec{r}_3\right|} = G\frac{m_1 m_3}{\left[\left(x_1 - x_3\right)^2 + \left(y_1 - y_3\right)^2\right]^{3/2}}\begin{pmatrix} x_1 - x_3 \\ y_1 - y_3 \end{pmatrix}.$$

Thus the total force acting on planet 1 is just

$$\vec{F}_{12} + \vec{F}_{13} = Gm_1\left\{m_2\frac{\vec{r}_1 - \vec{r}_2}{\left|\vec{r}_1 - \vec{r}_2\right|^3} + m_3\frac{\vec{r}_1 - \vec{r}_3}{\left|\vec{r}_1 - \vec{r}_3\right|^3}\right\}$$

$$= Gm_1\left\{\frac{m_2}{\left[\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2\right]^{3/2}}\begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \end{pmatrix} + \frac{m_3}{\left[\left(x_1 - x_3\right)^2 + \left(y_1 - y_3\right)^2\right]^{3/2}}\begin{pmatrix} x_1 - x_3 \\ y_1 - y_3 \end{pmatrix}\right\}$$

Now Newton's Law says that $\vec{F} = m\vec{a}$; since $\vec{a} = \vec{r}\,''$, this tells us that $\vec{F} = m\vec{r}\,''$ and thus the motion of planet 1 is governed by the equation

$$\vec{r}_1\,'' = G\left\{m_2\frac{\vec{r}_1 - \vec{r}_2}{\left|\vec{r}_1 - \vec{r}_2\right|^3} + m_3\frac{\vec{r}_1 - \vec{r}_3}{\left|\vec{r}_1 - \vec{r}_3\right|^3}\right\}$$

or equivalently

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}'' = G\left\{\frac{m_2}{\left[\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2\right]^{3/2}}\begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \end{pmatrix} + \frac{m_3}{\left[\left(x_1 - x_3\right)^2 + \left(y_1 - y_3\right)^2\right]^{3/2}}\begin{pmatrix} x_1 - x_3 \\ y_1 - y_3 \end{pmatrix}\right\}.$$

We can proceed similarly to find the equations for planet 2 and planet 3; they are

$$\vec{r}_2\,'' = G\left\{m_1\frac{\vec{r}_2 - \vec{r}_1}{\left|\vec{r}_2 - \vec{r}_1\right|^3} + m_3\frac{\vec{r}_2 - \vec{r}_3}{\left|\vec{r}_2 - \vec{r}_3\right|^3}\right\}$$

and

$$\vec{r}_3\,'' = G\left\{m_1\frac{\vec{r}_3 - \vec{r}_1}{\left|\vec{r}_3 - \vec{r}_1\right|^3} + m_2\frac{\vec{r}_3 - \vec{r}_2}{\left|\vec{r}_3 - \vec{r}_2\right|^3}\right\}.$$

Combining these, we obtain the following second-order system of differential equations

$$\begin{cases} \vec{r}_1'' = G\left\{ m_2 \dfrac{\vec{r}_1-\vec{r}_2}{\left|\vec{r}_1-\vec{r}_2\right|^3} + m_3 \dfrac{\vec{r}_1-\vec{r}_3}{\left|\vec{r}_1-\vec{r}_3\right|^3} \right\} \\ \vec{r}_2'' = G\left\{ m_1 \dfrac{\vec{r}_2-\vec{r}_1}{\left|\vec{r}_2-\vec{r}_1\right|^3} + m_3 \dfrac{\vec{r}_2-\vec{r}_3}{\left|\vec{r}_2-\vec{r}_3\right|^3} \right\} \\ \vec{r}_3'' = G\left\{ m_1 \dfrac{\vec{r}_3-\vec{r}_1}{\left|\vec{r}_3-\vec{r}_1\right|^3} + m_2 \dfrac{\vec{r}_3-\vec{r}_2}{\left|\vec{r}_3-\vec{r}_2\right|^3} \right\} \end{cases}$$

which can be written in terms of the functions x_i and y_i as

$$\begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \end{pmatrix}'' = G \begin{pmatrix} m_2 \dfrac{(x_1-x_2)}{\left[(x_1-x_2)^2+(y_1-y_2)^2\right]^{3/2}} + m_3 \dfrac{(x_1-x_3)}{\left[(x_1-x_3)^2+(y_1-y_3)^2\right]^{3/2}} \\ m_2 \dfrac{(y_1-y_2)}{\left[(x_1-x_2)^2+(y_1-y_2)^2\right]^{3/2}} + m_3 \dfrac{(y_1-y_3)}{\left[(x_1-x_3)^2+(y_1-y_3)^2\right]^{3/2}} \\ m_1 \dfrac{(x_2-x_1)}{\left[(x_2-x_1)^2+(y_2-y_1)^2\right]^{3/2}} + m_3 \dfrac{(x_2-x_3)}{\left[(x_2-x_3)^2+(y_2-y_3)^2\right]^{3/2}} \\ m_1 \dfrac{(y_2-y_1)}{\left[(x_2-x_1)^2+(y_2-y_1)^2\right]^{3/2}} + m_3 \dfrac{(y_2-y_3)}{\left[(x_2-x_3)^2+(y_2-y_3)^2\right]^{3/2}} \\ m_1 \dfrac{(x_3-x_1)}{\left[(x_3-x_1)^2+(y_3-y_1)^2\right]^{3/2}} + m_2 \dfrac{(x_3-x_2)}{\left[(x_3-x_2)^2+(y_3-y_2)^2\right]^{3/2}} \\ m_1 \dfrac{(y_3-y_1)}{\left[(x_3-x_1)^2+(y_3-y_1)^2\right]^{3/2}} + m_2 \dfrac{(y_3-y_2)}{\left[(x_3-x_2)^2+(y_3-y_2)^2\right]^{3/2}} \end{pmatrix}.$$

Because this is a second order system, we must convert it to a first order system before we can use the numerical techniques that we have learned. Let $u_1 = x_1'$ be the *x*-component of the velocity of planet 1, and let $v_1 = y_1'$ be the *y*-component of the velocity of planet 1; in fact, let $u_i = x_i'$ be the *x*-component of the velocity of planet i, and let $v_i = y_i'$ be the *y*-component of the velocity of planet i where i can be either 1, 2, or 3.

We then obtain the following system of ordinary differential equations

$$\begin{pmatrix} x_1 \\ y_1 \\ u_1 \\ v_1 \\ x_2 \\ y_2 \\ u_2 \\ v_2 \\ x_3 \\ y_3 \\ u_3 \\ v_3 \end{pmatrix}' = G \begin{pmatrix} u_1 \\ v_1 \\ m_2 \dfrac{(x_1 - x_2)}{\left[(x_1 - x_2)^2 + (y_1 - y_2)^2\right]^{3/2}} + m_3 \dfrac{(x_1 - x_3)}{\left[(x_1 - x_3)^2 + (y_1 - y_3)^2\right]^{3/2}} \\ m_2 \dfrac{(y_1 - y_2)}{\left[(x_1 - x_2)^2 + (y_1 - y_2)^2\right]^{3/2}} + m_3 \dfrac{(y_1 - y_3)}{\left[(x_1 - x_3)^2 + (y_1 - y_3)^2\right]^{3/2}} \\ u_2 \\ v_2 \\ m_1 \dfrac{(x_2 - x_1)}{\left[(x_2 - x_1)^2 + (y_2 - y_1)^2\right]^{3/2}} + m_3 \dfrac{(x_2 - x_3)}{\left[(x_2 - x_3)^2 + (y_2 - y_3)^2\right]^{3/2}} \\ m_1 \dfrac{(y_2 - y_1)}{\left[(x_2 - x_1)^2 + (y_2 - y_1)^2\right]^{3/2}} + m_3 \dfrac{(y_2 - y_3)}{\left[(x_2 - x_3)^2 + (y_2 - y_3)^2\right]^{3/2}} \\ u_3 \\ v_3 \\ m_1 \dfrac{(x_3 - x_1)}{\left[(x_3 - x_1)^2 + (y_3 - y_1)^2\right]^{3/2}} + m_2 \dfrac{(x_3 - x_2)}{\left[(x_3 - x_2)^2 + (y_3 - y_2)^2\right]^{3/2}} \\ m_1 \dfrac{(y_3 - y_1)}{\left[(x_3 - x_1)^2 + (y_3 - y_1)^2\right]^{3/2}} + m_2 \dfrac{(y_3 - y_2)}{\left[(x_3 - x_2)^2 + (y_3 - y_2)^2\right]^{3/2}} \end{pmatrix}.$$

To this system we must add initial conditions. In particular, we need to specify each of the twelve variables x_i, y_i, u_i and v_i for $i = 1,2,3$ at time $t = 0$. Physically, this requires us to know the positions and velocities for each of the three bodies when the simulation starts.

Project:

Write a C++ program that simulates the motion of three bodies under the influence of gravity.

As input, the program should take

- The initial positions and velocities of all of the bodies,
- The masses of all of the bodies,
- The gravitational constant of the universe,
- The time that the simulation should run, and
- The step size used in the simulation.

As output, the program should return

- A graphical representation of the motion of the bodies and
- The final positions and velocities of all of the bodies.

The program should be written using good object oriented programming techniques.

You are then to write up a technical report that answers the following questions:

1. What is the mathematical model of the problem?
2. What is the numerical method used to solve the problem?
3. What is the structure of your program?
4. Describe the range of possible behaviors shown by the system.

In particular, for question 4, consider the following questions:
Is it possible for the system to eject one of the bodies to infinity?
Is it possible for the system to return to its initial state?
Is it possible for the system to return to a state close to, but not the same as, its initial state?

When answering these questions, you must address the question of how the choice of step size affects the result.

Project: The Double Spring

Section 1: Introduction

Suppose that we have two springs attached to a mass at one end, with the other ends fixed at the points $(-1,0)$ and $(1,0)$.

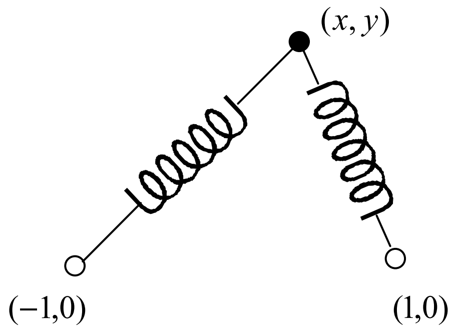


Figure 1: The double spring system

If we let go of the mass, it will be pushed and pulled around the plane by the springs. Our question is to determine the resulting motion of the mass.

Section 2: The Model

We begin by modeling a simple spring with one fixed end. Hooke's Law says that if the free end of a spring is stretched a distance d from its equilibrium position, then the spring exerts a force $F = -kd$ where k is a positive constant, called the spring constant. Equivalently, if x is the total distance the spring is stretched, then $F = -k(x-L)$ where L is the equilibrium length of the spring.

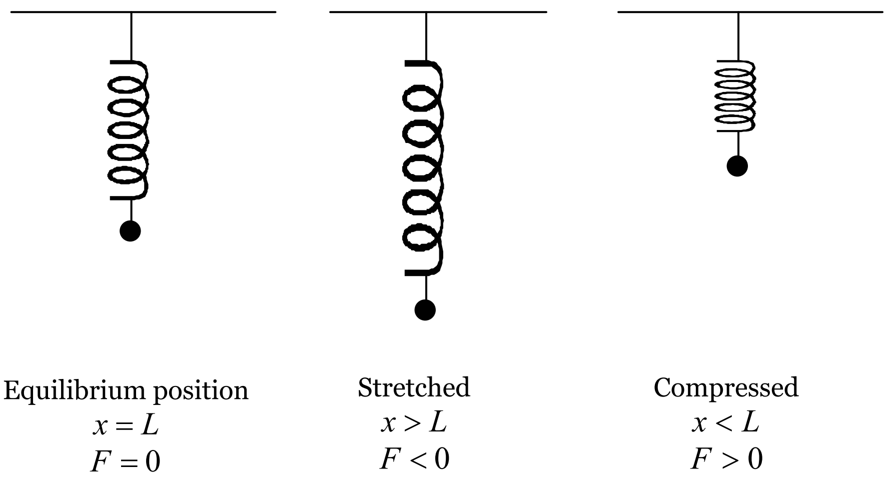


Figure 2: Illustration of Hooke's Law

If $x < L$, then the spring is compressed, and the free end is closer to the fixed point than its equilibrium position. In this case $F > 0$ and the spring pushes the free end back towards equilibrium. On the other hand, if $x > L$, then the spring has been stretched, the free end is pulled back toward equilibrium, and $F < 0$.

In the double spring system, let the moving object have mass m, and assume that the two springs are identical with spring constant k and equilibrium length L. To determine the motion of our object, we need to find the total force acting on it.

We begin by calculating the forces exerted by each spring. Let the vector from $(-1,0)$ to (x,y) be called $\vec{v}_1$; then $\vec{v}_1 = \begin{pmatrix} x+1 \\ y \end{pmatrix}$ and $\vec{v}_1$ has length $|\vec{v}_1| = \sqrt{(x+1)^2 + y^2}$. Thus the force exerted by that spring is $-k\left(\sqrt{(x+1)^2+y^2} - L\right)$. This force acts in the direction of $\vec{v}_1$; a vector of unit length in the same direction as $\vec{v}_1$ is

$$\frac{\vec{v}_1}{|\vec{v}_1|} = \frac{1}{\sqrt{(x+1)^2+y^2}}\begin{pmatrix} x+1 \\ y \end{pmatrix}.$$

Thus the total force acting on our moving mass from the spring attached to $(-1,0)$ is

$$\vec{F}_{(-1,0)} = k\left(\frac{L}{\sqrt{(x+1)^2+y^2}} - 1\right)\begin{pmatrix} x+1 \\ y \end{pmatrix}.$$

A similar argument tells us that the force acting on our moving mass from the spring attached to $(1,0)$ is

$$\vec{F}_{(1,0)} = k\left(\frac{L}{\sqrt{(x-1)^2+y^2}} - 1\right)\begin{pmatrix} x-1 \\ y \end{pmatrix}.$$

Our model of the double spring system is then

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \frac{k}{m}\left[\left(\frac{L}{\sqrt{(x-1)^2+y^2}} - 1\right)\begin{pmatrix} x-1 \\ y \end{pmatrix} + \left(\frac{L}{\sqrt{(x+1)^2+y^2}} - 1\right)\begin{pmatrix} x+1 \\ y \end{pmatrix}\right].$$

Because this is a second-order system, to solve it numerically we need to convert it to a first-order system. To do so, introduce the new variables $u = x'$ and $v = y'$. This gives us the system

$$\begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix}' = \begin{pmatrix} u \\ v \\ \frac{k}{m}\left(\frac{L}{\sqrt{(x-1)^2+y^2}}-1\right)(x-1)+\frac{k}{m}\left(\frac{L}{\sqrt{(x+1)^2+y^2}}-1\right)(x+1) \\ \frac{k}{m}\left(\frac{L}{\sqrt{(x-1)^2+y^2}}+\frac{L}{\sqrt{(x+1)^2+y^2}}-2\right)y \end{pmatrix}$$

To this system, we must add appropriate initial conditions, namely the position and velocity of the mass at time $t=0$.

Project:

Write a C++ program that simulates the motion of a double spring system.

As input, the program should take

- The initial position of the mass,
- The initial velocity of the mass,
- The spring constant of the two springs (assumed to be identical),
- The natural length of the two springs (assumed to be identical),
- The mass,
- The time that the simulation should run, and
- The step size used in the simulation.

As output, the program should return

- A graphical representation of the motion of the system and
- The final positions and velocities of the mass.

The program should be written using good object oriented programming techniques.

You are then to write up a technical report that answers the following questions:

1. If x_0=0.524, y_0=0.546, u_0=-0.036, v_0=-0.841, m=1.0, L=0.5, and k=50.0, what is the position of the mass at t=5.0? Is the final value of y positive or negative? How accurate is your answer?
2. Are there initial conditions for which the solution remains above the x-axis for all time?
3. The differential equation

$$\begin{cases} y' = f(t,y) \\ y(t_0) = y_0 \end{cases}$$

displays *sensitive dependence on initial conditions* at (t_0, y_0) if small changes in either t_0 or y_0 can produce significant changes in the solution. Does the system display sensitive dependence on initial conditions? If so, does this always happen, or is it true only for some data?

4. Suppose you apply a force $F(x)$ on an object at position x. The *work* done in moving the object from $x = a$ to $x = b$ is defined to be
$$W = \int_a^b F(x)\, dx .$$
What is the work done by a one-dimensional spring to move an object from equilibrium to the point x?
5. Consider
$$E = \tfrac{1}{2} m\left((x')^2 + (y')^2\right) - \tfrac{1}{2} k\left[\left(\sqrt{(x+1)^2 + y^2} - L\right)^2 + \left(\sqrt{(x-1)^2 + y^2} - L\right)^2\right].$$
Show that this quantity is conserved. What is its significance?
Hints:
   - By conserved, we mean that E is constant for all time. We can show this by proving $E' = 0$.
   - For the significance, consider #4, and recall that the kinetic energy of a body of mass m and moving with velocity v is $\tfrac{1}{2} mv^2$.

When answering these questions, you must address the question of how the choice of step size affects the result.

Sliders and Scrolling

Section 1: Introduction

Many programs use scroll bars to control various program functions. We shall learn how to integrate scroll bars into our code. As a demonstration, we shall write a short program that uses scroll bars to adjust the size and position of an ellipse drawn on the screen.

Section 2: The Skeleton

We shall begin by creating a simple dialog-based program called `Scrolling` with no About Box. In the main dialog, we shall remove the Cancel button and rename the OK button to Exit Program. From the Controls toolbar, we then select Slider.

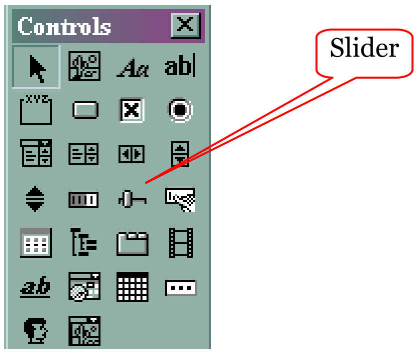
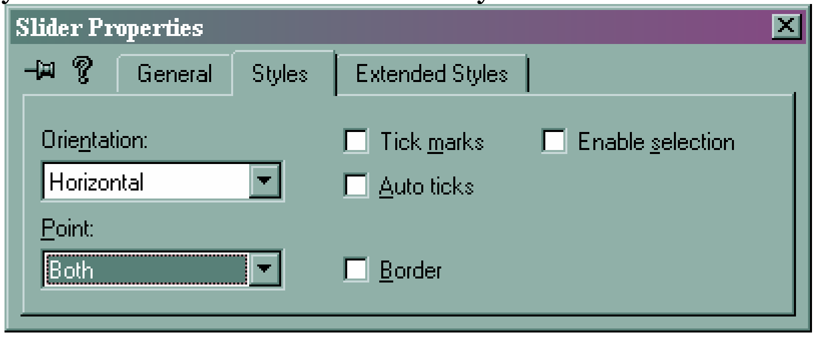


Figure 1: The slider entry in the Controls toolbar

We add four sliders. Two of these will be for adjusting the position of our ellipse on the screen, and two will be used to adjust its height and width. We would like to leave two of the sliders in their default, horizontal orientation, but we would like two of them to be oriented vertically. We can change the orientation of a slider by selecting it, right-clicking to get its properties menu. Its orientation is controlled by the Orientation control in the Styles tab. Note that when the



Orientation: Horizontal
Point: Both
☐ Tick marks
☐ Auto ticks
☐ Border
☐ Enable selection

Figure 2: The Slider Properties menu

slider orientation is changed, the actual size of the slider in the dialog box is left unchanged. We need to resize the slider so that it can be seen. At this point, we can also add tick marks and/or a border to our slider if we desire. We add appropriate ID's for each of our slider bars and each of our edit boxes. We call then IDD_SLIDER_VPOS, IDD_EDIT_VPOS, IDD_SLIDER_HPOS, IDD_EDIT_HPOS and so on. Finally, we add four edit boxes, one for each of our sliders. At this point, our dialog box should look something like Figure 3.

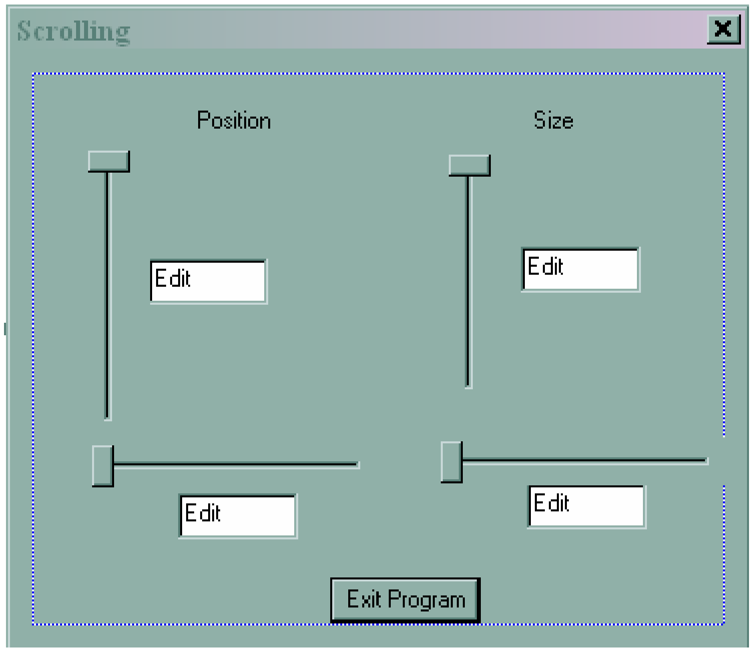


Figure 3: The controls

Section 3: The Class `CEllipse`

Because we want to use our controls to adjust the position and size of an ellipse, we create a second dialog box and a class for it, called `CEllipse`. This class has four private variables,

- `m_ptWindowCenter`
- `m_ptEllipseCenter`

of type CPoint, and

- `m_cxWidth`
- `m_cyHeight`

which are both integers. The first of these is to hold the center point of the dialog window; the second is the center point of the ellipse, while the last two contain the width and height of the ellipse.

Our `OnPaint()` method simply reads

```
void CEllipse::OnPaint()
{
      CPaintDC dc(this); // device context for painting
```

```
        // TODO: Add your message handler code here

        CBrush* OldBrush;
        CBrush YellowBrush;
        YellowBrush.CreateSolidBrush(RGB(255,255,0));
        OldBrush = dc.SelectObject(&YellowBrush);

        dc.Ellipse(m_ptEllipseCenter.x - m_cxWidth,
              m_ptEllipseCenter.y - m_cyHeight,
              m_ptEllipseCenter.x + m_cxWidth,
              m_ptEllipseCenter.y + m_cyHeight);

        dc.SelectObject(OldBrush);

        // Do not call CDialog::OnPaint() for painting messages
  }
```
Recall that we can edit the `OnPaint()` function by right-clicking on the `CEllipse` class, and choosing Add Windows Message Handler, then selecting `WM_PAINT`.

We need to initialize the variables in our program. Our first instinct might be to do this in the constructor, but this will not work. The reason for this is that we would like the `m_ptWindowCenter` to hold the actual center point of our window. However the class is created before it is assigned a window, thus this information is unknown at construct time. Instead, we would like to perform this initialization when the window is first created. We can do this by adding a handler to the window message `WM_CREATE` which is called when the window is created. We then add the following code

```
  int CEllipse::OnCreate(LPCREATESTRUCT lpCreateStruct)
  {
        if (CDialog::OnCreate(lpCreateStruct) == -1)
              return -1;

        // TODO: Add your specialized creation code here

        CRect CurrentRect;
        GetClientRect(CurrentRect);
        m_ptWindowCenter = CurrentRect.CenterPoint();
        m_ptEllipseCenter = m_ptWindowCenter;
        m_cxWidth = CurrentRect.Width()/2;
        m_cyHeight = CurrentRect.Height()/2;

        return 0;
  }
```

This gives us reasonable initial values for our parameters.

Finally, we need to create a public method to set the parameters for our ellipse; we use the following public function

```
  void CEllipse::SetData(int x, int y, int w, int h)
```

```
{
      m_cxWidth = w;
      m_cyHeight = h;
      m_ptEllipseCenter = CPoint(x,y) + m_ptWindowCenter;
      Invalidate();

}
```

Note that this function takes as input the offset from the center for our ellipse, rather than its absolute center.

Finally, we return to the class `CSrollingDlg` create a private variable of type `CEllipse` called `m_dlgEllipse`. In `OnInitDialog()` we add the code in the box below.

```
BOOL CScrollingDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
automatically
      //  when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon

      // TODO: Add extra initialization here

      m_dlgEllipse.Create(IDD_ELLIPSE);
      m_dlgEllipse.ShowWindow(SW_SHOW);

      return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

At this point, when our program runs, we should be presented with a dialog box like Figure 3 and a dialog box like Figure 4.

Section 4: Sliders

To use our slider bars, we need to associate them with a variable. This can be done in two different ways. We can associate an integer variable to a slider. The range for the integer variable is 0 to 100, and we read and set the value of the variable using `UpdateData()`.

As the second option, we can also assign a control variable to a slider. In the class wizard, when associating the variable, change the Category from Value to Control. The variable type for a slider control is `CSliderCtrl`. We can set the range for the values for `CSliderCtrl` by using the `SetRange()` method, which takes integer values for the low and the high end of the range. We can set the value for the `CSliderCtrl` by using `SetPos()` method. There is no need to call

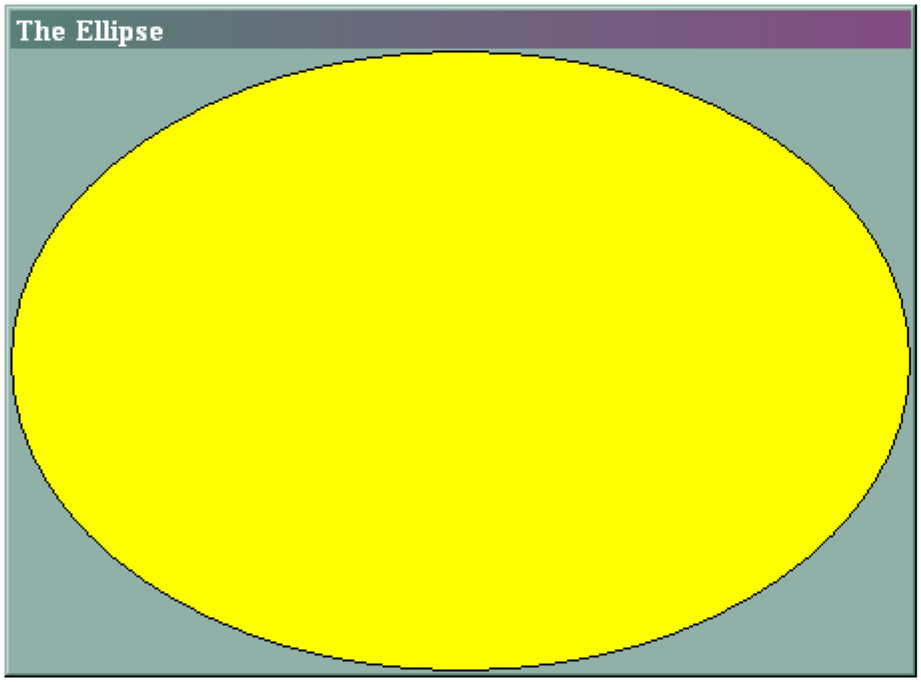


Figure 4: The result of our `CEllipse` class.

`UpdateData()` when using `SetPos()`. We determine the value for the `CSLiderCtrl` by using the `GetPos()` method. For more information on `CSliderCtrl`, see MDSN.

In this example, we shall use controls. We associate the variables `m_ctrlHPosSlider`, `m_ctrlHSizeSlider`, `m_ctrlVPosSlider`, and `m_ctrlVSizeSlider` to the appropriate sliders; we also associate the integers `m_iHPos`, `m_iHSize`, `m_iVPos` and `m_iVSize` to the corresponding edit boxes.

We shall initialize these values in the OnInitDialog() method of our class CEllipseDlg. We add the following code to set default values for the edit boxes and scroll bars; we then call the SetData(int x, int y, int w, int h) function from the class CEllipse to synchronize the data.

```
BOOL CScrollingDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
automatically
      //  when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon

      // TODO: Add extra initialization here

      m_dlgEllipse.Create(IDD_ELLIPSE);
      m_dlgEllipse.ShowWindow(SW_SHOW);
      m_ctrlHPosSlider.SetRange(-100,100,TRUE);
      m_ctrlVPosSlider.SetRange(-100,100,TRUE);

      m_iHPos = 0;
      m_iVPos = 0;
      m_iHSize = 50;
      m_iVSize = 50;

      UpdateData(FALSE);

      m_ctrlHPosSlider.SetPos(m_iHPos);
      m_ctrlVPosSlider.SetPos(m_iVPos);
      m_ctrlHSizeSlider.SetPos(m_iHSize);
      m_ctrlVSizeSlider.SetPos(m_iVSize);

      m_dlgEllipse.SetData(m_iHPos,m_iVPos,m_iHSize,m_iVSize);

      return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

The new code is indicated in the box. The default range for the parameter in a slider is from 0 to 100. The first two commands change the range to –100 to 100. Note that the `SetPos(int)` functions do not requires an `UpdateData(BOOL)` command, but that the position of the slider bars is updated immediately. Notice also that the new code is placed after the code that generates the dialog box that shows the ellipse; this is so that the `m_dlgEllipse.SetData` command can be executed.

Section 5: Scrolling

To use the scroll bars, we would like to update the size or position of the ellipse whenever the position of one of the slider bars is changed. Whenever the position of a slider control is changed, the program receives a windows message. For horizontal slider bars and scroll bars, it receives a WM_HSCROLL message, while for a vertical slider bar or scroll bar, it receives a WM_VSCROLL message.

The same message is sent regardless of the source of the scroll message ; however the message handler receives a pointer to the scroll bar which generated the message. Indeed, when the handler to a WM_HSCROLL message is first added to the code, it has the form

```
void CEllipse::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
      // TODO: Add your message handler code here and/or call
default

      CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

The first parameter received describes the type of scrolling that generated the message. Allowable values for this parameter include

• `SB_LEFT`	The user has scrolled to the far left.
• `SB_RIGHT`	The user has scrolled to the far right.
• `SB_LINELEFT`	The user has scrolled to the left.
• `SB_LINERIGHT`	The user has scrolled to the right
• `SB_THUMBPOSITION`	The user has scrolled to an absolute position indicated by `nPos`.
• `SB_THUMBTRACK`	The user has dragged the scrollbar to the position indicated by `nPos`.

The nPos parameter is only used by `SB_THUMBPOSITION` and `SB_THUMBTACK`. The last parameter is the pointer to the scroll bar that generated the message. The last parameter is a pointer to the `CScrollBar` that generated the message.

`CScrollBar` is a class that can be used to dynamically change the properties of a scroll bar. This class is derived from the more general `CWnd` class. It is used to tell us which scroll bar or slider bar generated the current scroll message. We can do this by using the `GetDlgCtrlID()` function which `CScrollBar` inherits from `CWnd`. This method will return the ID of the window.

We can now add the code to the `OnHScroll` method.

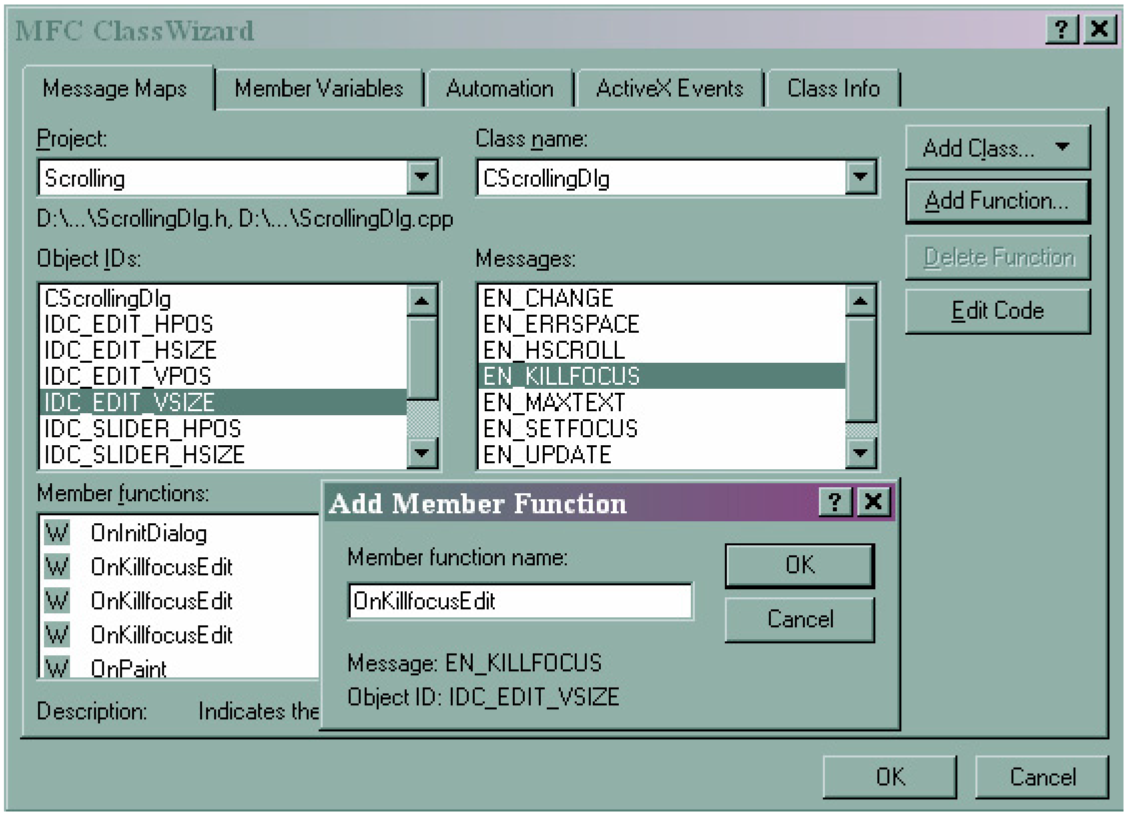
```
void CScrollingDlg::OnHScroll(UINT nSBCode, UINT nPos,
CScrollBar* pScrollBar)
{
      // TODO: Add your message handler code here and/or call
default

      switch(pScrollBar->GetDlgCtrlID())
      {
      case IDC_SLIDER_HPOS:
            m_iHPos = m_ctrlHPosSlider.GetPos();
            UpdateData(FALSE);
            m_dlgEllipse.SetData(m_iHPos,m_iVPos,
                  m_iHSize,m_iVSize);
            break;
      case IDC_SLIDER_HSIZE:
            m_iHSize = m_ctrlHSizeSlider.GetPos();
            UpdateData(FALSE);
            m_dlgEllipse.SetData(m_iHPos,m_iVPos,
                  m_iHSize,m_iVSize);
            break;
      }


      CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

The new material is boxed. We use the `GetDlgCtrlID()` function to determine the ID of the slider that generated the scroll message. We then store the position of the slider in the appropriate integer variable, and update the edit box. This way the value in the edit box will remain synchronized with the slider. We then call the `SetData` function from `m_dlgEllipse` to change the size and/or position of the ellipse. The last thing that is done is to call the `OnHScroll` method for the `CDialog` which handles all other scroll messages.

We also write a similar function for `WM_VSCROLL`. The primary difference is that the parameter `nSBCode` can take on different values:

• `SB_BOTTOM`	The user has scrolled to the bottom.
• `SB_TOP`	The user has scrolled to the top.
• `SB_LINEDOWN`	The user has scrolled down one line
• `SB_LINEUP`	The user has scrolled up one line
• `SB_THUMBPOSITION`	The user has scrolled to an absolute position indicated by `nPos`.
• `SB_THUMBTRACK`	The user has dragged the scrollbar to the position indicated by `nPos`.

At this point, our code is nearly complete. All that remains for us to do is to update the appropriate slider and the ellipse whenever the value in the edit box is changed. Although there is a message that is sent whenever the value of an edit box changes, called `EN_CHANGE`, it is inappropriate for this purpose. This is



Project: Scrolling

Class name: CScrollingDlg

D:\...\ScrollingDlg.h, D:\...\ScrollingDlg.cpp

Object IDs:
CScrollingDlg
IDC_EDIT_HPOS
IDC_EDIT_HSIZE
IDC_EDIT_VPOS
IDC_EDIT_VSIZE
IDC_SLIDER_HPOS
IDC_SLIDER_HSIZE

Messages:
EN_CHANGE
EN_ERRSPACE
EN_HSCROLL
EN_KILLFOCUS
EN_MAXTEXT
EN_SETFOCUS
EN_UPDATE

Member functions:
W OnInitDialog
W OnKillfocusEdit
W OnKillfocusEdit
W OnKillfocusEdit
W OnPaint

Description: Indicates the

Add Member Function

Member function name: OnKillfocusEdit

Message: EN_KILLFOCUS
Object ID: IDC_EDIT_VSIZE

Figure 5: Adding the message map for EN_KILLFOCUS

because it will also check to see that the value in the box is an integer, and if the user deletes the current value before typing in a new value, the program will display an error dialog asking that an integer value be added to the edit box. Instead, we shall use the `EN_KILLFOCUS` message, which is sent when an edit box loses the input focus. This can occur whenever the user clicks on another portion of the program or another edit box.

To add this message handler, we start the Class Wizard (Ctrl+W), then select the Message Maps tab. Select the Object ID, and appropriate message, then choose Add Function. The Class Wizard provides a default name for the function.

In our example, we shall use the same function `OnKillFocusEdit` for the `EN_KILLFOCUS` message for all of our edit boxes. We use the following code for this function.

```
void CScrollingDlg::OnKillfocusEdit()
{
      // TODO: Add your control notification handler code here

      UpdateData(TRUE);

      m_ctrlHPosSlider.SetPos(m_iHPos);
      m_ctrlVPosSlider.SetPos(m_iVPos);
      m_ctrlHSizeSlider.SetPos(m_iHSize);
      m_ctrlVSizeSlider.SetPos(m_iVSize);

      m_dlgEllipse.SetData(m_iHPos,m_iVPos,m_iHSize,m_iVSize);

}
```

Assignments:

1. What essential feature is missing in the `OnKillFocusEdit()` method described in the text?
2. Write a program that draws a circle on the screen. Your program should have three slider bars that can be used to adjust the color of the resulting circle. The slider bars will control the amount of red, green, and blue that are used to color the circle.

Dynamic Memory Allocation and Graphs

Section 1: Introduction

In C++, memory for data objects can be allocated and deallocated dynamically, i.e. during the execution time. This feature is very desirable when we are dealing with the applications containing data structures whose size is not known during the compilation time. Rather than statically reserving memory for the maximum possible size of the data structure, we can delay the memory allocation till run time during which we can obtain the size of the data structure dynamically. In this section we briefly describe the dynamic memory allocation and introduce the pointer data type that is used to point to a memory location.

Section 2: Dynamic Memory Allocation

A variable of pointer data type contains an address of data object. Though pointers are memory addresses, we have to define the type of data object to which a pointer variable points. Following is the declaration of a pointer variable called `intPtr` that points to a single integer.

```
int* intPtr;
```

C++ provides two operators for the allocation and deallocation of the memory. The `new` operator is used to allocate memory and returns its address; the `delete` operator deallocates the memory. The memory for the objects created dynamically is allocated on the heap and it will not deallocated automatically as object goes out of scope and ceases to exit. It is the programmer's responsibility to deallocate the memory to prevent memory leaks.

Dynamically creating a variable is a two-step process. First, you declare a pointer to the type of variable; then you use the new function to allocate the memory. Consider the following code fragment:

```
int n;
n=10;
double* x;            // This creates the pointer
x = new double[n]     // This creates an array of size n
```

Dynamic memory allocation can be used in other contexts; the following code fragment dynamically creates a single integer, and gives it the value 5.

```
int* n;
n = new int (5);
```

The process of dynamically creating a variable can be shortened to one line. Consider the following:

```
int* n = new int (5);
```

This is equivalent to the two-line construction above. Note however, that this line cannot be placed in a header file.

Variables created dynamically exist independently of the scope in which they were created. In particular, these variables exist until they are destroyed manually. This is done using the delete operator. To delete an array, use `delete[]`; otherwise use `delete`. Consider the following code fragment:

```
double* x;
x = new double[10]
int* n = new int (5);

delete[] x;
delete n;
```

Recall that when an instance of a class goes out of scope, its destructor is called. This is an excellent place to put any needed delete operations.

Section 3: Graphs

Often in mathematics and modeling it is useful to see the graph of a function. In this section we shall write a short program that displays the graph of the function $f(t) = A\cos(\omega t + \phi)$ where the values of A, ω, and ϕ will be entered by the user. The user will also enter the range of t values for which the graph will be shown. This is a simple example that requires dynamic allocation of memory, as the number of points is an input variable. Moreover, we show how to use the `PolyLine` method defined in `CDC` class. The `PolyLine` method takes an array of points and the number of points and draws a series of line segments by connecting all the points in the array.

The program for this example contains three main components:

1. A user interface that allows the user to input the data.
2. An output window that displays the graph for the function.
3. A processing engine that, based on the number of input points and the size of the output window (both obtained dynamically), calculates the points for the given function and invokes the OnPaint function of the output window to be drawn.

Due to the simplicity of the graph, we have not used a separate class to calculate the points; rather we have included it in the message handler of the Graph button, on the user interface window.

User Interface

Create a dialog-based project and call it Dynamic. Next, insert a dialog box. In case you forgot how to do it, click on the ResourceView tab in the workspace pane, right click on Dialog option and select Insert. Remove the "TODO" text and the Cancel button. Right click on the OK button and select Properties option to change the caption of OK button to Exit Program. Next add 5 edit box controls, corresponding to A, ω, ϕ, number of points and end time.

Set the ID property of edit box controls to: `IDC_A`, `IDC_OMEGA`, `IDC_PHI`, `IDC_N` and `IDC_ENDTIME`. Add 5 Static Text controls and change their Captions as to End Time, A, Phi, Omega and Number of Points. Next add one Command Button Control and change its ID and caption to IDC_BUTTON_GRAPH and Graph. Figure 1, depicts the dialog box with all the controls.

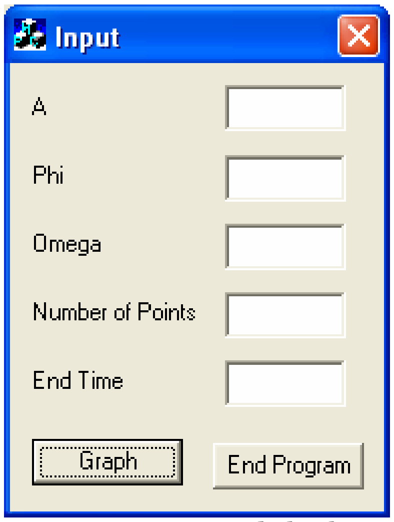
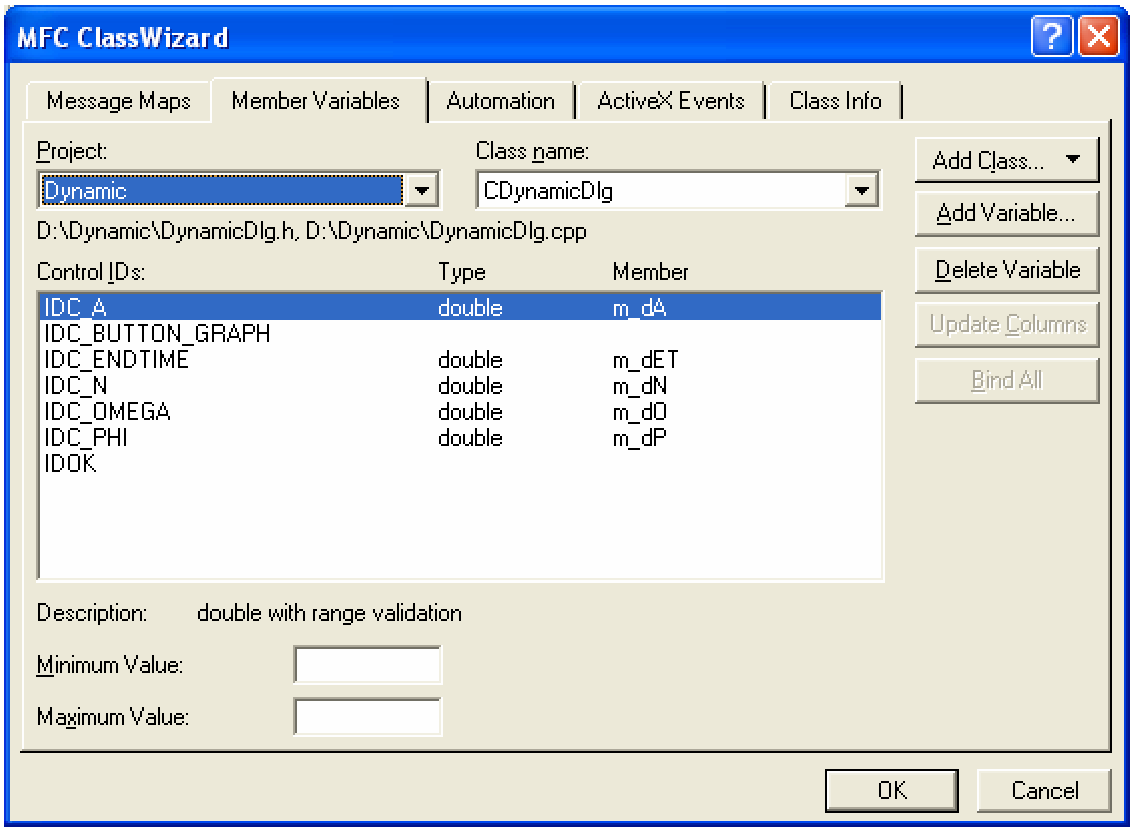


Figure 1: Our main dialog box

Now we need to attach variable names and functionality to these controls before we can use them in our application. Invoke the ClassWizard by entering CTRL+W, click on the Member Variables tab, select a Control Id from the list of Control IDs and click on the Add Variable option. Attach a variable name and a data type to each control ID. The variable names and types for the Edit controls are shown in Figure 2. To initialize these variables, click on the constructor function for the `CDynamicDlg` class, and modify the initial values to those shown below:

```
CDynamicDlg::CDynamicDlg(CWnd* pParent /*=NULL*/)
      : CDialog(CDynamicDlg::IDD, pParent)
{
      //{{AFX_DATA_INIT(CDynamicDlg)
      m_dA = 1;
      m_dET =10;
      m_dN = 100;
      m_dO = 1;
```



Project: Dynamic

Class name: CDynamicDlg

D:\Dynamic\DynamicDlg.h, D:\Dynamic\DynamicDlg.cpp

Control IDs:	Type	Member
IDC_A	double	m_dA
IDC_BUTTON_GRAPH		
IDC_ENDTIME	double	m_dET
IDC_N	double	m_dN
IDC_OMEGA	double	m_dO
IDC_PHI	double	m_dP
IDOK		

Description: double with range validation

Minimum Value:

Maximum Value:

Figure 2: Variable names

```
        m_dP = 0.0;

        //}}AFX_DATA_INIT
        // Note that LoadIcon does not require a subsequent
        // DestroyIcon in Win32
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

Next, we want to insert the second dialog box into the project and change its caption to Graph and its ID to `IDD_GRAPH`. Remove the "TODO" text, Cancel button and OK button. This dialog will be used to display the graph for our function. Create a class for this dialog and call it `CGraph`. Use the ClassWizard and add the following private member variables:

```
int m_iN;                //Number of points
CRect m_rectWin;         //width of the graph window
CPoint * m_ptPoints;     //pointer to the array of points to be
                         //plotted
bool m_bInit;            //a flag indicating whether memory for
                         //the array of points have been
                         //allocated.
```

In the constructor function, we set the initial value of `m_bInit` flag to `false` and `m_ptPoints` to `NULL`.

```
CGraph::CGraph(CWnd* pParent /*=NULL*/)
      : CDialog(CGraph::IDD, pParent)
{
      //{{AFX_DATA_INIT(CGraph)
            // NOTE: the ClassWizard will add member
            // initialization here
      //}}AFX_DATA_INIT
      m_bInit = false;
      m_ptPoints = NULL;
}
```

Since we are dealing with dynamic memory allocation, we need to add a destructor function to deallocate the memory when the object goes out of scope. Below is the destructor function.

```
CGraph::~CGraph()
{
      if (m_bInit)
            delete [] m_ptPoints;
}
```

We add the following public function that calculates the coordinates of the points to be plotted. This function is called when the Graph button on the Input dialog is selected. Since during the execution of the program, we may draw more than one graph by changing the input parameters, the first thing this function does, deallocates the memory dynamically created for the array of points, if exists. Next, creates the array of points for the given number of points and after setting the flag to true it calculates the *x* and *y* coordinates for the points to be plotted, using the dimensions of the current Graph window.

```
void CGraph::CreateGraph(double *x, double *y, int N, double T)
{
      if (m_bInit)
            delete [] m_ptPoints;

      m_ptPoints = new CPoint[N];
      m_bInit = true;
      m_iN = N;

      for (int i =0 ; i <N; i++)
      {
            m_ptPoints[i].x=  (int)(x[i]* m_rectWin.Width()/ T);
            m_ptPoints[i].y = (int)((double)(m_rectWin.Height())
                              * (y[i]-1)/(-2*1));
      }

      Invalidate();
      OnPaint();
}
```

Next, to get the current widow size and plot the graph we need to add and edit the message handlers associated with `WM_INITDIALOG` and `WM_PAINT` messages. To do this, right click on the `CGraph` class name in Class View pane and select Add Windows Message Handler option. Then select `WM_INITDIALOG` from the New Windows Messages/Events pane and click Add and Edit button on the right.

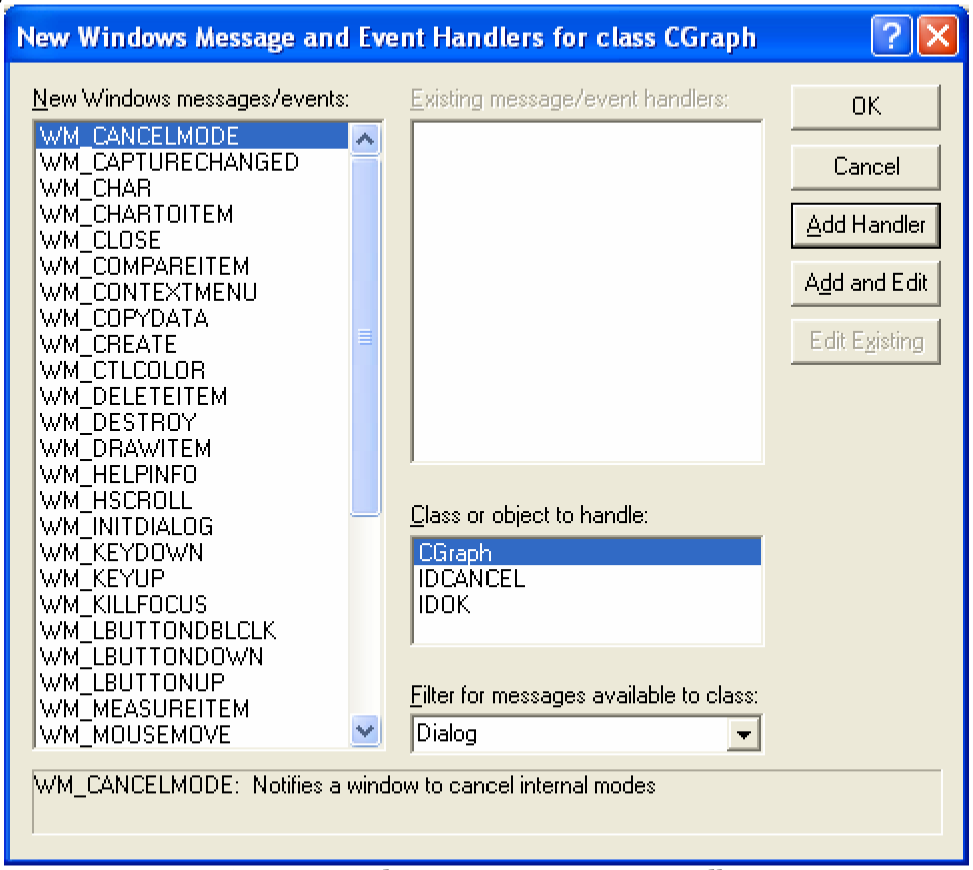


Figure 3: Windows Event & Message Handlers

First we modify the `OnInitDialog` message handler. In order to get the width and height of the Graph window in the above for loop, we need to call the `GetClientRect` function to initialize these properties. Click on the `OnInitDialog` function name in the ClassView tab and add the code shown in the box to this function.

```
BOOL CGraph::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	GetClientRect(m_rectWin);
	return TRUE;  // return TRUE unless you set the focus to a
	              //  control
	              // EXCEPTION: OCX Property Pages should
	              //  return FALSE
}
```

Next we make the modifications to `OnPaint` message handler. If we do have a set of points to plot, we call `PolyLine`, which uses the current pen, to draw the graph.

```
void CGraph::OnPaint()
{
      CPaintDC dc(this); // device context for painting

      // TODO: Add your message handler code here
      if (m_bInit)
            dc.Polyline(m_ptPoints,m_iN);
      // Do not call CDialog::OnPaint() for painting messages
}
```

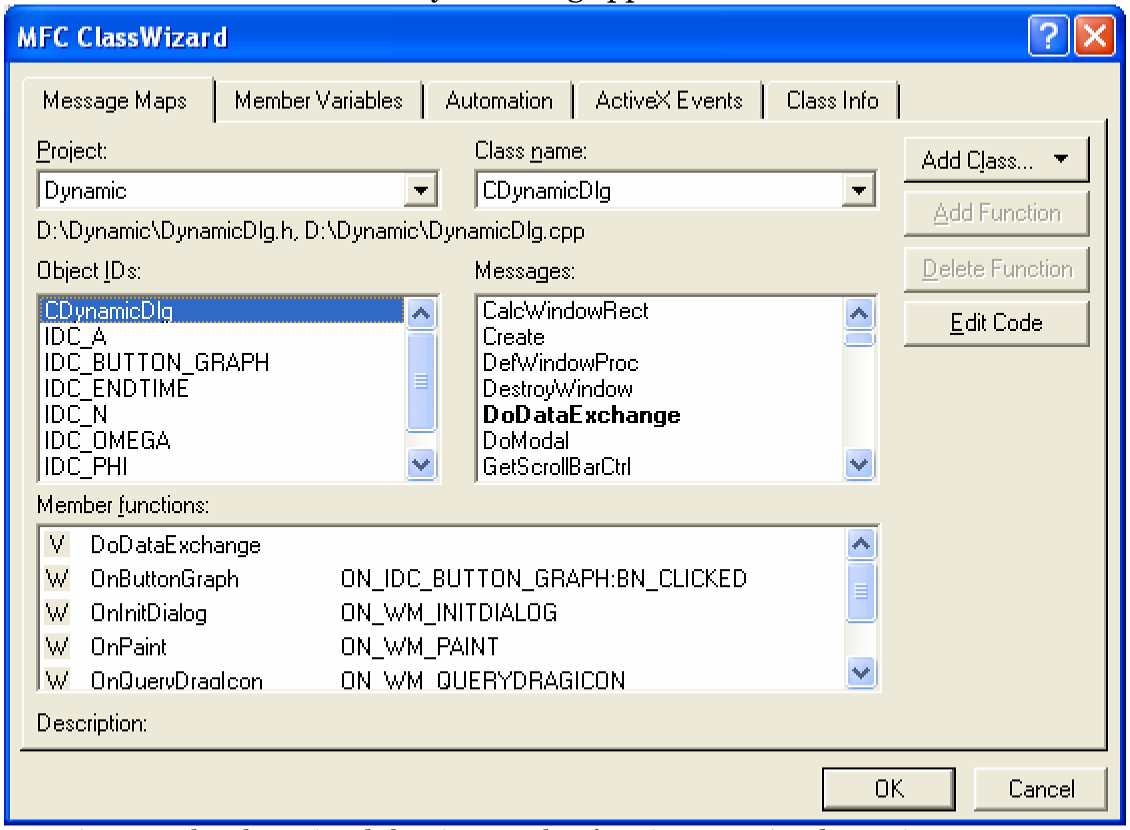
So far, we have explained how these dialog boxes are created and operate. In order to connect the connect them together, we add a variable of type `CGraph`, called `m_dlgGraph` to the `CDynamicDlg` class and invoke `Create` and `ShowWindow` to display the Input Dialog. The `OnButtonGraph` is called to display the graph for the initial values provided by the programmer. If you wish to initialize all values to zeroes, you can remove the `OnButtonGraph` call. Or you may leave the initial values and click the Graph button to draw the graph when the program is executed. Modify the `OnInitDialog` function of `CDynamic` class by adding the code segments shown below.

```
BOOL CDynamicDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
automatically
      //  when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon

      // TODO: Add extra initialization here
      m_dlgGraph.Create(IDD_GRAPH);
      m_dlgGraph.ShowWindow(SW_SHOW);
      OnButtonGraph();         // remove if you do not provide
                              // initial data.
      return TRUE;  // return TRUE  unless you set the focus to a
                    // control
}
```

Below is the message handler for the Graph button on the Input dialog box. The `OnButtonGraph` function gets the input data, evaluates the function $f(t) = A\cos(\omega t + \phi)$ and stores the values in two dynamically created arrays x and y which are passed to `CreateGraph` function to be plotted.

```
void CDynamicDlg::OnButtonGraph()
{
      // TODO: Add your control notification handler code here
      UpdateData(true);

      double* x;
      double* y;

      x = new double [m_dN];
      y = new double [m_dN];
      for (int i =0 ; i <m_dN; i++)
      {
            x[i] = ((double)(i)/(double)(m_dN)* m_dET);
            y[i]= m_dA*cos(m_dO*x[i]+m_dP);
      }
    m_dlgGraph.CreateGraph(x,y,m_dN, m_dET);
}
```

To attach this function to the Graph button, invoke the ClassWizard and select Message Maps. Under the Object IDs window, select `IDC_BUTTON_GRAPH` and double click on `BN_CLICKED` in the Messages window. A dialog box will be opened and allow you to enter the name of member function you want to attach to `BN_CLICKED` message. Figure 4 depicts the member functions in `CDynamic` class that are associated with messages. A screen shot of the output when the program is successfully compiled and executed is shown in Figure 5. Remember to include `<math.h>` in the CDynamicDlg.cpp file.



Project: Dynamic

Class name: CDynamicDlg

D:\Dynamic\DynamicDlg.h, D:\Dynamic\DynamicDlg.cpp

Object IDs:

CDynamicDlg
IDC_A
IDC_BUTTON_GRAPH
IDC_ENDTIME
IDC_N
IDC_OMEGA
IDC_PHI

Messages:

CalcWindowRect
Create
DefWindowProc
DestroyWindow
DoDataExchange
DoModal
GetScrollBarCtrl

Member functions:

```
V  DoDataExchange
W  OnButtonGraph        ON_IDC_BUTTON_GRAPH:BN_CLICKED
W  OnInitDialog         ON_WM_INITDIALOG
W  OnPaint              ON_WM_PAINT
W  OnQueryDragIcon      ON_WM_QUERYDRAGICON
```

Description:

Figure 4: The ClassWizard showing member functions associated to various messages

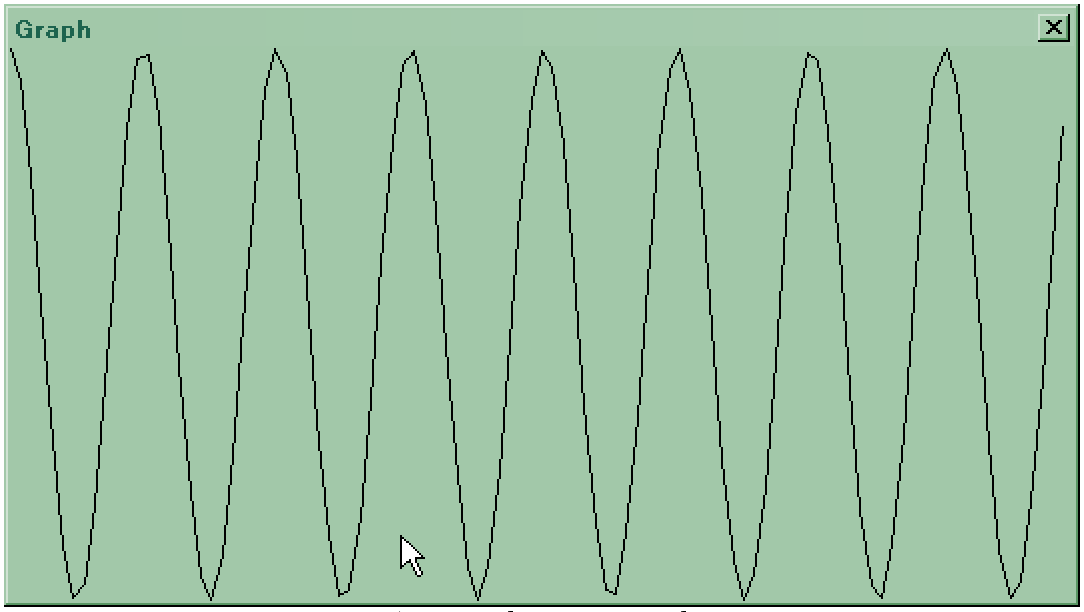


Figure 5: The output graph.

Project: The Resonant Filter

Section 1: Introduction

How can you tune a radio? More generally, suppose that we have an electrical signal with components at a number of different frequencies. How can we construct a circuit that will pass some frequencies, but block others?

In this project, we shall describe and model a resonant filter. We will then create a simulation of the resonant filter, and shall use it to determine the components of an unknown input signal.

Section 2: Current and Voltage

Let us begin with a brief discussion of electronics. Electric charge is measured in units of coulombs (C). One coulomb is equivalent to the charge on 6.25×10^{18} protons; or equivalently the negative of the charge on 6.25×10^{18} electrons. The charge on a proton is positive, while the charge on an electron is negative.

Electrical current is the amount of electrical charge crossing a point in a unit of time. It is measured in amps (A), with 1 A = 1 C/s. Voltage is the amount of energy required to move a unit of electrical charge; it is measured in volts (V), with 1 V = 1 J/C. Recall that the joule (J) is a unit of energy; 1 J = 1 N m = 1 kg $\mathrm{m}^2/\mathrm{s}^2$.

Ohm's Law

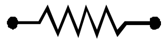
If two points connected by an electrical conductor (say a wire) have different voltages then an electrical current will flow from the point with higher voltage to the point with lower voltage. If the voltage difference is V, then the amount of current I that flows satisfies

$$V = IR$$

where R is a characteristic of the conductor, called the resistance. This statement is *Ohm's Law*. The units of resistance are Ohms (Ω), with 1 Ω = 1 V/A.

Wires used to conduct electricity are designed so that their resistance is as low as possible. In fact, we shall assume that the resistance in a wire has been lowered all the way to zero. As a consequence, any two points on a wire with no circuit element between them must be at the same voltage.

However, when designing a circuit, there are times when we deliberately want to impede the flow of electricity. This is done by using a circuit element called a resistor. In a circuit diagram, the symbol for a resistor is



We often refer to a resistor by its resistance R.

Consider the circuit in Figure 1, where the resistor has resistance R. Suppose that the voltage difference between A and E is known and called V_{in}. The current I flowing between B and F then satisfies $V_{in} = IR$. The voltage V_{out}, which is the difference in voltage between D and G, is the same as $V_{in} = IR$.

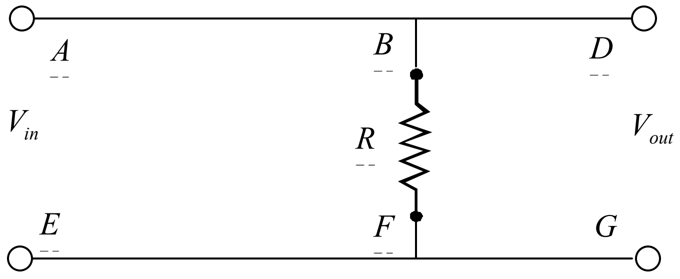


Figure 1: A circuit with a single resistor

Section 3: Circuit Elements

Capacitors

A *capacitor* is a device for storing charge. A simple capacitor consists of a pair of parallel flat plates. Charge flowing towards one plate accumulates there. This creates an electrical field in the region between the plates, which causes the opposite charge to accumulate on the opposite plate. The two plates in a capacitor can be at different voltages, but because they are not connected, Ohm's Law does not apply.

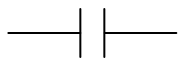
If the voltage difference between the two plates in the capacitor remains constant, then no current flows across the capacitor. If the voltage on one side changes, then one plate begins to accumulate charge. The accumulation of charge on the opposite plate is equivalent to current flowing across the capacitor. In an ideal capacitor, we have the relationship

$$I = C\frac{dV}{dt},$$

where I is the current that flows across the capacitor, V is the voltage difference across the capacitor, and C is a constant of the capacitor, called is capacitance. The unit of capacitance is the Farad (F) with 1 F = 1 C/V. Sometimes we write instead that

$$V = \frac{1}{C}\int I\,dt.$$

The symbol of a capacitor is



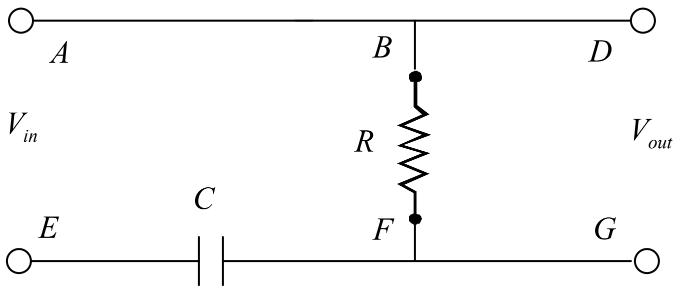


Figure 2: Circuit with a resistor and a capacitor

Consider the circuit in Figure 2. Let us suppose for simplicity that no current flows from D to B to F to G, but that there is a current I that flows from A to B to F to E. Then the voltage drop from B to F is IR, while the voltage drop from F to E is $V = \frac{1}{C}\int I\,dt$. Combining these, we see that

$$\begin{aligned} V_{in} &= IR + \frac{1}{C}\int I\,dt, \\ V_{out} &= IR. \end{aligned}$$

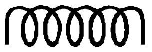
Inductors

A simple *inductor* is formed by a tightly coiled piece of wire. Moving current in the wire creates a magnetic field in the region between the coils. Changing the current in the wire changes the size of the induced magnetic field; however, a changing magnetic field creates a voltage difference. In an ideal inductor, we have the relationship

$$V = L\frac{dI}{dt},$$

where V is the voltage across the inductor, I is the current that flows across the inductor, and L is a constant, called the inductance. The unit of inductance is the Henry (H), with 1 H = 1 V s/A = 1 Ω s.

The symbol of an inductor is



Consider the circuit in Figure 3, and suppose as before that no current flows from D to B to F to G. What can we say about V_{in} and V_{out}?

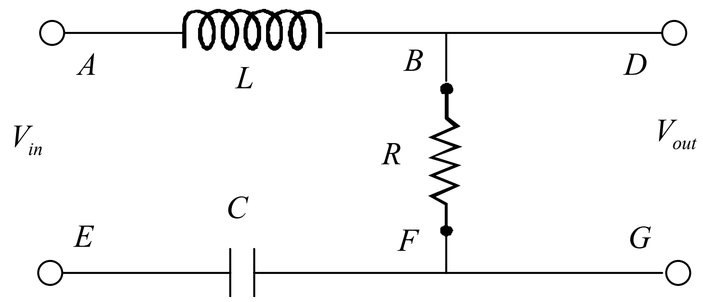


Figure 3: A circuit with a resistor, capacitor, and inductor

Let I be the current that flows from A to B to F to E. Then the voltage drop from B to F is IR, the voltage drop from F to E is $\frac{1}{C}\int I \; dt$, and the voltage drop from A to B is $L\frac{dI}{dt}$. Combining these, we see that

$$\begin{aligned} V_{in} &= \frac{1}{C}\int I\, dt + RI + LI' \\ V_{out} &= RI \end{aligned} \tag{1}$$

The circuit in figure 3 is called a resonant filter. The reason for this is that if a sinusoidal voltage V_{in} is applied, then the amplitude of the output voltage depends strongly on the frequency of V_{in}. Certain frequencies will be passed through the circuit only slightly changed, while other frequencies will be strongly attenuated.

To demonstrate this effect, we need to understand the behavior of solutions to (1). We begin by rewriting (1) as

$$\frac{L}{R}V''_{out} + V'_{out} + \frac{1}{RC}V_{out} = V'_{in}. \tag{2}$$

To analyze this equation, we begin by analyzing general second-order, constant-coefficient differential equations.

Section 4: Properties of Second Order Differential Equations

Consider the initial-value problem

$$\begin{cases} Ay'' + By' + Cy = f \\ \qquad\qquad y(0) = y_0 \\ \qquad\qquad y'(0) = y_1 \end{cases} \tag{3}$$

where $A > 0$, $B > 0$, and $C > 0$. What are the analytic properties of the solutions to this problem?

The Homogeneous Case

To analyze (3), begin by assuming that $f = 0$ for simplicity. This is called the *homogeneous* problem. We look for solutions of the form $y = e^{kt}$ where k is a constant to be determined later. Direct substitution shows that k satisfies

$$Ak^2 + Bk + C = 0.$$

Thus

$$k = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

There are three possibilities depending on the value of the discriminant $B^2 - 4AC$.

Case 1: $B^2 - 4AC > 0$. In this case, both values of k are negative, and we find two different solutions, both of which decay exponentially as $t \to \infty$.

Case 2: $B^2 - 4AC < 0$. In this case, the values of k form a complex conjugate pair, with

$$k = -\frac{B}{2A} \pm i\frac{\sqrt{4AC - B^2}}{2A} = \alpha \pm i\beta .$$

Here we use the notation $i = \sqrt{-1}$. To understand the solutions in this case, we need to know something of the behavior of the function $y = e^{(\alpha + i\beta)t}$. For simplicity, we begin by examining $g(x) = e^{ix}$. Recall the Taylor series expansion for e^x, to wit,

$$e^x = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \cdots$$

Substituting, we see that

$$e^{ix} = 1 + ix - \frac{1}{2!}x^2 - \frac{i}{3!}x^3 + \frac{1}{4!}x^4 + \cdots$$

Now if we collect all of the real terms and all of the imaginary terms separately, we find

$$e^{ix} = \left(1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots\right) + i\left(x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \cdots\right)$$

which we immediately recognize as

$$e^{ix} = \cos x + i \sin x .$$

This famous result is called DeMoivre's Formula.

Thus, the solution in this case is $e^{(\alpha \pm i\beta)t} = e^{\alpha t}\left(\cos\beta t \pm i \sin\beta t\right)$. Now the sum, difference, or a constant multiple of two solutions to $Ay'' + By' + Cy = 0$ is also a solution, so we obtain the two real solutions $y = e^{\alpha t}\cos\beta t$ and $y = e^{\alpha t}\sin\beta t$. Since $\alpha = -\dfrac{B}{2A} < 0$, we see that both of our solutions decay exponentially as $t \to \infty$.

Case 3: $B^2 - 4AC = 0$. In this case, our method finds only one solution $y = e^{\frac{-B}{2A}t}$. The second solution has the form $y = te^{\frac{-B}{2A}t}$, which can be verified by substitution. In this case however, both solutions decay exponentially as $t \to \infty$.

The Nonhomogeneous Case

Now let us examine what happens in the general case. We want to solve the initial-value problem

$$\begin{cases} Ay'' + By' + Cy = f \\ y(0) = y_0 \\ y'(0) = y_1 \end{cases} \tag{3}$$

Suppose that we have already determined a function that satisfies

$$Ay'' + By' + Cy = f,$$

and let us call it y_p. Further, let us suppose that we have two solutions y_1 and y_2 of the homogeneous problem $Ay'' + By' + Cy = 0$.

The sum, difference, or constant multiple of a solution to the homogeneous problem $Ay'' + By' + Cy = 0$ is also a solution of the homogeneous problem. Thus, for any pair of numbers λ_1 and λ_2, the combination

$$\lambda_1 y_1 + \lambda_2 y_2 + y_p$$

is a solution of

$$Ay'' + By' + Cy = f.$$

Thus, to find a solution to our initial value problem, it is sufficient to find the values of λ_1 and λ_2 so that

$$\begin{aligned} \lambda_1 y_1(0) + \lambda_2 y_2(0) + y_p(0) &= y_0 \\ \lambda_1 y_1'(0) + \lambda_2 y_2'(0) + y_p'(0) &= y_1 \end{aligned}.$$

In our case, this is always possible. Because we know that $y_1(t)$ and $y_2(t)$ tend to zero as $t \to \infty$, the combination $\lambda_1 y_1 + \lambda_2 y_2$ is called the transient solution, while y_p is called the steady state solution.

Section 5: The Gain of the Filter

Our model for the resonant filter is

$$\frac{L}{R} V''_{out} + V'_{out} + \frac{1}{RC} V_{out} = V'_{in}.$$

Our analysis has shown that the solution of any initial-value problem for this equation consists of a transient solution that decays to zero exponentially fast to ensure that the initial conditions are satisfied. This is added to the steady-state solution, which is any particular solution of the equation. Our analysis however, did not describe any method to obtain this particular solution.

Rather than find the particular solution for an arbitrary choice of V_{in}, let us instead look for solutions where the input voltage is the sum of sines and cosines. Thanks to exercise 2, we know that if V_{in} is the sum of two terms, then we can analyze each term separately. Thus we can assume that V_{in} is simply a sinusoid at just one frequency. Further, the sum of sinusoids

$$\lambda_1 \cos(\omega t + \phi_1) + \lambda_2 \sin(\omega t + \phi_2)$$

can always be rewritten as

$$\alpha_1 e^{i\omega t} + \alpha_2 e^{-i\omega t}$$

[Exercise 5] so we can assume that V_{in} is the sum of complex exponentials. Thus, we begin our analysis by supposing that

$$V_{in} = e^{i\omega t}.$$

With this assumption, we need to find a particular solution $V(t)$ to the equation

$$\frac{L}{R}V'' + V' + \frac{1}{RC}V = i\omega e^{i\omega t}.$$

This solution will be our steady-state solution. Because of the form of V_{in}, we look for a particular solution with the form $V = Ge^{i\omega t}$. Substituting, we find that

$$-\frac{L}{R}\omega^2 G + i\omega G + \frac{1}{RC}G = i\omega.$$

Solving for G, we discover that

$$G = \frac{i\omega}{\left(\frac{1}{RC} - \frac{L}{R}\omega^2\right) + i\omega}.$$

Thus, if V_{in} is a sinusoid, we conclude that V_{out} is also a sinusoid, and that the ratio V_{out}/V_{in} is given by

$$\frac{V_{out}}{V_{in}} = |G|.$$

The quantity $|G|$ is called the gain of the filter. It is the ratio of the amplitude of the output signal to the input signal. If $|G| \approx 0$, then the signal is strongly attenuated, while if $|G| \approx 1$, then the size of the output signal is roughly the same size as the input signal. It is important to note that the gain of the filter depends on the frequency of the input signal. For example, if $R = 1\Omega$, $L = 25\,\text{mH}$, and $C = 1\,\mu\text{F}$, Figure 4 gives the gain $|G|$. From this figure, we see that if $\omega \approx 6325$, then the output signal has roughly the same amplitude as the input signal. On the other hand, for other values of ω, the signal is strongly attenuated; if $\omega \approx 6100$, then the output signal is roughly 10% of the size of the input signal.

To use our resonant filter to tune to a particular frequency, we want to find the value of ω that maximizes $|G|$; this is the preferred frequency of the filter. To find the maximum value, it is algebraically simpler to find the minimum of $|1/G|$. Now

$$\frac{1}{G} = 1 + \frac{1}{i\omega}\left(\frac{1}{RC} - \frac{L}{R}\omega^2\right) = 1 - \frac{i}{\omega}\left(\frac{1}{RC} - \frac{L}{R}\omega^2\right)$$

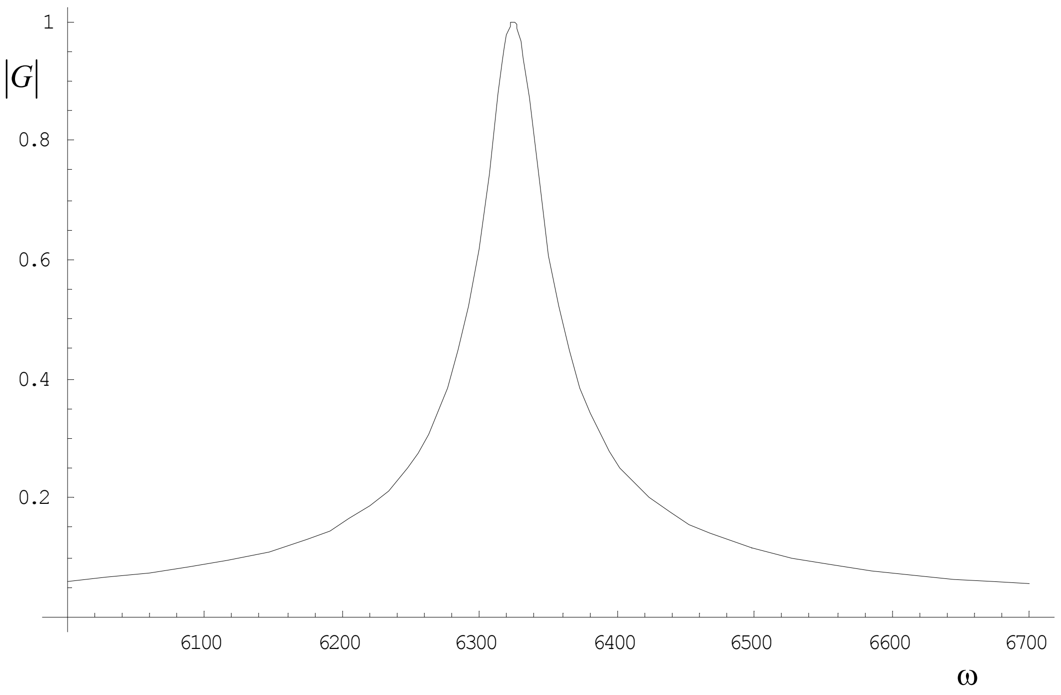


Figure 4: The gain $|G|$ when $R = 1\,\Omega$, $L = 25\,\text{mH}$, and $C = 1\,\mu\text{F}$,

so that

$$\left|\frac{1}{G}\right| = \sqrt{1 + \left[\frac{1}{\omega}\left(\frac{1}{RC} - \frac{L}{R}\omega^2\right)\right]^2}$$
$$= \sqrt{1 + \frac{L^2}{R^2\omega^2}\left(\omega^2 - \frac{1}{LC}\right)^2}.$$

Thus

$$|G| = \frac{1}{\sqrt{1 + \frac{L^2}{R^2\omega^2}\left(\omega^2 - \frac{1}{LC}\right)^2}},$$

so we conclude that the maximum value of $|G|$ is 1 which occurs when $\omega = \dfrac{1}{\sqrt{LC}}$.

Note that if $L = 25\,\text{mH}$, and $C = 1\,\mu\text{F}$, then $\dfrac{1}{\sqrt{LC}} = 2000\sqrt{10} \approx 6325$.

Assignments

1. Suppose that $y_1(t)$ and $y_2(t)$ satisfy the equation $Ay''(t) + By'(t) + Cy(t) = 0$. Show that, for any numbers λ_1 and λ_2, that the function $\lambda_1 y_1(t) + \lambda_2 y_2(t)$ also satisfies the equation.

2. Suppose that $y_1(t)$ satisfies the equation $Ay''(t)+By'(t)+Cy(t)=f_1(t)$ and $y_2(t)$ satisfies the equation $Ay''(t)+By'(t)+Cy(t)=f_2(t)$. Show that, for any numbers λ_1 and λ_2, that the function $\lambda_1 y_1(t)+\lambda_2 y_2(t)$ also satisfies the equation $Ay''(t)+By'(t)+Cy(t)=\lambda_1 f_1(t)+\lambda_2 f_2(t)$

3. Prove that the function $y = te^{\frac{-B}{2A}t}$ decays to zero as $t \to \infty$.

4. The function $y(t)$ decays exponentially to zero if there are numbers $\lambda > 0$, $\alpha > 0$ and $T > 0$ so that $|y(t)| \le \lambda e^{-\alpha t}$ for all $t > T$. Prove that $y = te^{\frac{-B}{2A}t}$ decays exponentially to zero. [Hint: Choose $\alpha < \frac{B}{2A}$, and consider $\lim_{t\to\infty} \frac{y(t)}{e^{-\alpha t}}$.]

5. Show that, for any choices of the parameters λ_1, λ_2, ϕ_1, and ϕ_2, the function

$$\lambda_1 \cos(\omega t + \phi_1) + \lambda_2 \sin(\omega t + \phi_2)$$

can be written in the form

$$\alpha_1 e^{i\omega t} + \alpha_2 e^{-i\omega t}$$

for some choices of α_1 and α_2.

6. For the resonant filter, determine the corresponding homogeneous solution. What is the condition on L, R, and C that determines if the transient solution has an oscillatory component?

7. Why does the transient solution have that name? Why does the steady-state solution have that name?

Project

Write a C++ program that simulates the behavior of an LRC resonant filter. In particular, it should simulate the response of the resonant filter to the unknown signal provided in class.

As input, the program should take

- The inductance,
- The capacitance,
- The resistance, and
- The end time.

As output, the program should return

- A graph of the initial signal,
- A graph of the output signal, and
- The preferred frequency of the circuit.

The program should let the user enter the inductance, capacitance, resistance, and end time using either slider bars or by directly entering the value in text boxes.

The input data is provided in the form of a class CInputData. It has methods to retrieve the data, and its derivative. It contains the values of the input signal sampled at 1000 equally spaced points in the interval [0,1]. The initial data is periodic with period 1, and so this class can be used to retrieve the data for all times.

The input data is the sum of two sinusoids of different and unknown frequencies. You are to use your simulation to determine these unknown frequencies. You should carefully explain your reasoning.

The program should be written using good object oriented programming techniques.

Adaptive Methods for Ordinary Differential Equations

The Runge-Kutta-Fehlberg Method

Section 1: Introduction

The Runge-Kutta method provides a good estimate of the solution; however, it does not also estimate the error in its approximation at the same time. The Runge-Kutta-Fehlberg method provides both an estimate for the solution and at the same time that it provides an estimate for the error.

Section 2: The Method

To solve the initial-value problem

$$\begin{cases} y' = f(t,y) \\ y(t_0) = y_0 \end{cases}$$

we need to compute a sequence of times t_i and approximations y_i so that $y_i \approx y(t_i)$. In all of the methods we have learned so far, we would choose a single step size h that we would use for the entire calculation. There is a problem with this approach however. The differential equation may need a very small value of h for an accurate approximation, but only for a short period of time. If we used one value of h for the entire computation, either we waste time using a small step size for those portions of the computations when it is not required, or we lose accuracy by choosing a larger step size. The solution is to allow the step size h to vary as the computation progresses.

How can we determine how to change the step size in a computation? The best way would be to use an estimate for the local error in the computation. What is the local error? Suppose that we are calculating a numerical approximation to the solution of the initial value problem

$$\begin{cases} y' = f(t,y) \\ y(t_0) = y_0 \end{cases}$$

and that we have calculated the approximation (t_i, y_i). We then calculate the next approximation (t_{i+1}, y_{i+1}). The *global error* at t_{i+1} is the difference $y_{i+1} - y(t_{i+1})$; in general this is difficult to calculate. To define the local error, suppose that we have the same differential equation, but change the initial data to agree with the approximation (t_i, y_i). In particular, let $\tilde{y}(t)$ be the solution of

$$\begin{cases} \tilde{y}' = f(t, \tilde{y}) \\ \tilde{y}(t_i) = y_i \end{cases}.$$

Then the *local error* is $y_{i+1} - \tilde{y}(t_{i+1})$. Although not as useful as the global error, the main advantage of the local error is that it is computable.

In an adaptive method, we will vary the step size for each iteration to ensure that the local error is less than some prescribed tolerance at each step. This does not guarantee that the global error is small however.

How can we estimate the local error? One way is to use two different numerical methods of two different orders to approximate the solution of the equation; in many common cases an estimate of the local error can then be derived. The difficulty with this approach is the necessity of doing twice as much work. It would be nice if we could perform one set of calculations to obtain two different numerical approximations.

One method with this characteristic is the *Runge-Kutta-Fehlberg* method. It is a pair consisting of a fourth-order method and a fifth-order method that allows us to estimate the local error. It works as follows. Suppose that the approximation (t_i, y_i) has already been computed. We then compute the following:

$$k_1 = h \cdot f(t_i, y_i)$$

$$k_2 = h \cdot f\left(t_i + \frac{1}{4}h, y_i + \frac{1}{4}k_1\right)$$

$$k_3 = h \cdot f\left(t_i + \frac{3}{8}h, y_i + \frac{3}{32}k_1 + \frac{9}{32}k_2\right)$$

$$k_4 = h \cdot f\left(t_i + \frac{12}{13}h, y_i + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3\right)$$

$$k_5 = h \cdot f\left(t_i + h, y_i + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right)$$

$$k_6 = h \cdot f\left(t_i + \frac{1}{2}h, y_i - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right)$$

Our approximation for the solution is

$$y_{i+1} = y_i + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5. \tag{1}$$

Further, if we calculate the approximation

$$\tilde{y}_{i+1} = y_i + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 \tag{2}$$

Then the local error in the approximation is at most

$$|y_{i+1} - \tilde{y}_{i+1}| = \left| \frac{1}{360}k_1 - \frac{128}{4275}k_3 - \frac{2197}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6 \right|.$$

The result is a fourth-order method; however this method requires six function evaluations per step rather than the four in the traditional fourth-order Runge-Kutta method. This extra expense is offset however, by the error estimates which we can use to create an adaptive method.

Section 3: The Algorithm

In the following algorithm, we shall use the Runge-Kutta-Fehlberg method to compute each step, and the associated local error. If the local error is sufficiently small, we shall keep the results of that step, and adjust the step size for the next step. If the local error is too large however, we shall drop the results of the step, and repeat the process with a smaller step size.

As input, we have a maximum step size (hmax), a minimum step size (hmin) and a tolerance (TOL). Suppose that we have already calculated the data $(t_0, y_0), (t_1, y_1), \ldots, (t_i, y_i)$. We calculate (t_{i+1}, y_{i+1}) as follows. Use the current step size, and use (1) and (2) to calculate y_{i+1} and $\tilde{y}_{i+1}$. We then set

$$R = \frac{1}{h}|y_{i+1} - \tilde{y}_{i+1}| = \frac{1}{h}\left| \frac{1}{360}k_1 - \frac{128}{4275}k_3 - \frac{2197}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6 \right|. \tag{3}$$

If $R \le TOL$, then we accept the approximation for y_{i+1}, and we set $t_{i+1} = t_i + h$. If not, we shall toss out this approximation.

Next we calculate the factor $\delta = 0.84(TOL/R)^{1/4}$. This is our stretch factor to adjust the step size. If $\delta \le 0.1$, we replace δ by 0.1; if $\delta \ge 4$, we replace δ by 4. We then replace h by δh. If h is larger than hmax, we replace h by hmax. If h is less than hmin, we stop the program with an error saying that the desired tolerance could not be maintained.

Section 4: Systems

The Runge-Kutta-Fehlberg algorithm also works on systems; the modifications here are precisely the same modifications that were used for the fourth-order Runge-Kutta method. We can create an adaptive method in the same fashion as we did above; the only difference is in the computation of the quantity R from (3).

In the case of a system of n equations, we know that y_i and k_i are vectors with n components. Suppose that the components of y_i are $y_{i,1}$, $y_{i,2}$, ..., $y_{i,n}$ and that the components of k_i are $k_{i,1}$, $k_{i,2}$, ..., $k_{i,n}$. Then in place of (3) we calculate

$$R_1 = \frac{1}{h}\left|y_{i+1,1} - \tilde{y}_{i+1,1}\right| = \frac{1}{h}\left|\frac{1}{360}k_{1,1} - \frac{128}{4275}k_{3,1} - \frac{2197}{75240}k_{4,1} + \frac{1}{50}k_{5,1} + \frac{2}{55}k_{6,1}\right|$$
$$R_2 = \frac{1}{h}\left|y_{i+1,2} - \tilde{y}_{i+1,2}\right| = \frac{1}{h}\left|\frac{1}{360}k_{1,2} - \frac{128}{4275}k_{3,2} - \frac{2197}{75240}k_{4,2} + \frac{1}{50}k_{5,2} + \frac{2}{55}k_{6,2}\right|$$
$$\vdots$$
$$R_n = \frac{1}{h}\left|y_{i+1,n} - \tilde{y}_{i+1,n}\right| = \frac{1}{h}\left|\frac{1}{360}k_{1,n} - \frac{128}{4275}k_{3,n} - \frac{2197}{75240}k_{4,n} + \frac{1}{50}k_{5,n} + \frac{2}{55}k_{6,n}\right|$$

In place of R from (3), we then use

$$R = \max\left\{R_1, R_2, \ldots, R_n\right\}.$$

Assignment

1. Consider the initial-value problem

$$\begin{cases} y' = t + \sqrt{y^4 + 1} \\ y(0) = 0 \end{cases}.$$

Write a computer program that takes as input a step size, a minimum step size, a maximum step size, a tolerance, and a value of t. Your program should then calculate the Runge-Kutta-Fehlberg approximation to $y(t)$.

2. Consider the initial-value problem

$$\begin{cases} y''(t) + y(t) = \sin t \\ y(0) = 0 \\ y'(0) = 1 \end{cases}.$$

Write a computer program that takes as input a step size, a minimum step size, a maximum step size, a tolerance, and a value of t. Your program should then calculate the Runge-Kutta-Fehlberg approximation to $y(t)$. Compare your result to the exact solution $y(t) = \frac{3}{2}\sin t - \frac{t}{2}\cos t$ for different values of the step size h.

Project: Dynamics of HIV

Section 1: Introduction

We shall model the spread of HIV in the body using a simple one-compartment model to obtain a system of differential equations for the number of virus particles, the number of infected T cells, and the number of uninfected T cells. We shall use the model to examine the effect of RT inhibitor therapy.

Section 2: The Simple Model

Let us begin with a simple model of the concentration V of HIV virus particles in the blood. There are two main factors governing the number of virus particles present. First is the growth rate- this is the rate at which new virus particles are produced. This is an unknown function that we call P. Second is the clearance rate. The body attempts to clear itself of the virus; we shall assume that the rate at which is does so is proportional to the number of virus particles already present. This then gives us the simple model

$$\frac{dV}{dt} = P - cV .$$

The constant c is called the clearance rate; it too is unknown.

One immediate consequence of this model is that if we can force $P \approx 0$, by using an anti-viral drug, for example, then the concentration of the virus in the blood will drop exponentially. Indeed, if $P = 0$, then the model becomes

$$\frac{dV}{dt} = -cV$$

which has the solution

$$V = V_0 e^{-ct}$$

where V_0 is the virus concentration at time $t = 0$; this solution can be verified by substitution. Studied have been undertaken where patients were given anti-viral drugs and the concentration of the virus in their blood was measured. The virus concentration did drop exponentially, at least for a short period of time, at the researchers were able to obtain an estimate for c of 2.1 ± 0.4 days.

Section 3: White Blood Cells

Though helpful, the previous model is too simplistic to accurately represent the dynamics of HIV. The virus propagates by infecting white blood cells, and this simple model takes no notice of them. Thus, to create a more accurate model, we begin by examining white blood cells.

The primary target for HIV in the blood are white blood cells; in particular the CD4+ T cell. After becoming infected, such cells can produce new virus particles. The dynamics of CD4$^+$ T cells in the body are not well-understood. However, in the absence of HIV, there are three primary factors governing the growth rate of the number of CD4+ T cells in the body. First is the rate at which they are produced in the body in, for example, the thymus. We shall assume that this is a constant s. Next is the rate at which CD4+ T cells die. We shall assume that the death rate is proportional to the number of CD4+ T cells present; thus if we let T represent the concentration of CD4+ T cells in the body, then this term has the form $-d_T T$, where d_T is the death rate. The last major term comes from the fact that CD4+ T cells subdivide on their own. This effect is called the *proliferation* rate. The precise form for this term is unknown, however there is some evidence for a density dependent proliferation term. We shall model proliferation by a term of the form

$$pT\left(1-\frac{T}{T_{\max}}\right)$$

where $T_{\max}$ is the maximum sustainable number of CD4+ T cells in the body.

This type of term occurs often in population dynamics, and is called a *logistic* term. To understand it, let us first suppose that $T << T_{\max}$ In this case

$$pT\left(1-\frac{T}{T_{\max}}\right) \approx pT .$$

This then predicts a growth rate for the CD4+ T cell population that is proportional to the number of cells already present. On the other hand, for $T \approx T_{\max}$, we have

$$pT\left(1-\frac{T}{T_{\max}}\right) \approx 0$$

so that our growth rate slows as T approaches $T_{\max}$. Finally we note that if $T < T_{\max}$, then

$$pT\left(1-\frac{T}{T_{\max}}\right) > 0$$

so the component of the growth rate due to proliferation is positive, while if $T > T_{\max}$

$$pT\left(1-\frac{T}{T_{\max}}\right) < 0$$

and the component of the growth rate due to proliferation is negative.

Combining these features, we obtain the following model for the CD4+ T cell concentration in the absence of HIV

$$\frac{dT}{dt} = s + pT\left(1 - \frac{T}{T_{\max}}\right) - d_T T .$$

For reasonableness, we must have $s < d_T T_{\max}$. This ensures that the number of T cells is decreasing when we have reached the maximum concentration. This equation has a well-defined stable steady state

$$\bar{T} = \frac{T_{\max}}{2p}\left[p - d_T + \sqrt{(p - d_T)^2 + \frac{4sp}{T_{\max}}}\right].$$

Section 4: Interaction of HIV and White Blood Cells

In the presence of HIV, CD4+ T cells become infected. How can we model this?

Let T^* be the number of infected T cells. Assume that the rate at which T cells become infected is proportional to the number of T cells and the number of virus particles present. This is called a *mass-action* term, and it is reasonable if the virus and the T cells are well mixed, as we might expect in the blood stream. In this case, the dynamics of the uninfected CD4+ T cell concentration can be modeled by

$$\frac{dT}{dt} = s + pT\left(1 - \frac{T}{T_{\max}}\right) - d_T T - kVT$$

where the term $-kVT$ accounts for those CD4+ T cells that become infected.

We shall assume that infected T cells are produced only when the virus enters an uninfected T cell; thus we can model the concentration T^* of infected CD4+ T cells by

$$\frac{dT^*}{dt} = kVT - \delta T^* .$$

Here δ represents the death rate of infected T cells, which may be different that the death rate for uninfected T cells. We do not model the possibility of an infected T cell directly infecting another T cell.

At this point, we can return to the virus concentration V. If we assume that an infected T cell produces N new virus particles during its lifetime, then we can use the model $P = N\delta T^*$ and obtain the equation

$$\frac{dV}{dt} = N\delta T^* - cV .$$

This is a one-compartment model, meaning that we assume all of the virus particles and T cells are present in the blood. This is not accurate, because virus particles are actually present throughout the body, and the bulk of the T cells are concentrated in lymphoid tissue. However it is expected that blood concentrations reflect the state of the body as a whole.

The proliferation term $pT\left(1-\frac{T}{T_{\max}}\right)$ in our model does not account for the infected T cells; in fact a more reasonable model might be to use the term $pT\left(1-\frac{T+T^*}{T_{\max}}\right)$. However, in general we have $T^* \ll T$ so we can neglect the impact of the infected cells.

We have ignored loss of virus due to the infection of a T cell; this would result in a term $-kTV$ to the equation for dV/dt. However, measurements show that this effect is small compared to the clearance effect. If T is roughly constant, this can also be modeled by modifying the constant $c^* = c + kT$.

The biological mechanism for clearing the virus from the blood is unknown.

Section 5: Drug Therapy

A RT inhibitor is a drug that can be used on HIV infected patients. RT is an enzyme that is essential for HIV infection; without it a virus particle can enter a cell, but is unable to infect it. A perfect RT inhibitor will prevent HIV from infecting T cells, so our model becomes

$$\begin{aligned} \frac{dT}{dt} &= s + pT\left(1-\frac{T}{T_{\max}}\right) - d_T T \\ \frac{dT^*}{dt} &= -\delta T^* \\ \frac{dV}{dt} &= N\delta T^* - cV \end{aligned} \qquad .$$

Unfortunately, RT inhibitors are not foolproof. A more accurate model might be

$$\begin{aligned} \frac{dT}{dt} &= s + pT\left(1-\frac{T}{T_{\max}}\right) - d_T T - (1-\eta)kVT \\ \frac{dT^*}{dt} &= (1-\eta)kVT - \delta T^* \\ \frac{dV}{dt} &= N\delta T^* - cV \end{aligned}$$

where η is the effectiveness of the RT inhibitor. Here $\eta = 1$ corresponds to a perfect inhibitor, while $\eta = 0$ corresponds to no effect.

Assignments

1. Show that

$$T(t) = \bar{T} = \frac{T_{\max}}{2p}\left[p - d_T + \sqrt{(p - d_T)^2 + \frac{4sp}{T_{\max}}}\right]$$

is a constant solution of the differential equation

$$\frac{dT}{dt} = s + pT\left(1 - \frac{T}{T_{\max}}\right) - d_T T$$

modeling the CD4+ T cell concentration in the absence of HIV.

2. Show that the steady-state solution from question 1 is stable. In particular, suppose that $T \approx \bar{T}$. Show that if $T > \bar{T}$, then $\frac{dT}{dt} < 0$ and that if $T < \bar{T}$ then $\frac{dT}{dt} > 0$. Explain why this implies that $\bar{T}$ is a stable solution.

Project

Write a C++ program that simulates the spread of HIV through the body. Include the effect of a RT inhibitor.

Scenario 1

We shall suppose initially that the patient has virus in the body, but that no T cells have yet been infected.

Representative values for the parameters are given below. Units are microliters (μL) and days.

- T - the number of T cells. Initial value is $T = 1000$ μL^{-1}.
- T^* - the number of infected T cells. Initial value is $T^* = 0$ μL^{-1}.
- V - the number of virus particles. Initial value is $V = 10^{-3}$ μL^{-1}.
- s - the rate at which new T cells are generated in the body. $s = 10$ day^{-1}μL^{-1}.
- $T_{\max}$ - the maximum number of T cells in the body. $T_{\max} = 1500$ μL^{-1}.
- d_T - the natural death rate of uninfected T cells. $d_T = 0.02$ day^{-1}.
- δ - the death rate of infected T cells. $\delta = 0.24$ day^{-1}.
- c - the clearance rate of the virus. $c = 2.4$ day^{-1}.
- k - the infection rate. $k = 2.4 \times 10^{-5}$ day^{-1}.
- N - the number of virus particles produced during the lifetime of an infected T cell. $N = 1200$.

- p - the rate of proliferation of T cells. $p = 0.03$.

Answer the following questions.

1. What is the short-term behavior of the system?

2. Use your simulation to show that T, T^*, and V tend to constants as $t \to \infty$; say $T \to T_\infty$, $T^* \to T_\infty^*$, and $V \to V_\infty$. Estimate these values.

3. Explain analytically why $T_\infty = \frac{c}{Nk}$, $V_\infty = \frac{sN}{c} + \frac{p - d_T}{k} - \frac{pc}{Nk^2 T_{\max}}$, and $T_\infty^* = \frac{c}{N\delta} V_\infty$.

4. Does your simulation bear this out?

Scenario 2

Now let us analyze the effect of an RT inhibitor. Modify the preceding values as follows.

Set

- $T = 83.33\ \mu\text{L}^{-1}$
- $V_0 = 5347\ \mu\text{L}^{-1}$
- $T^* = 44.56\ \mu\text{L}^{-1}$

Answer the following questions.

1. Use your simulation to show that there is a critical value of the effectiveness parameter η, say η_{crit}, so that if $\eta > \eta_{crit}$, then $T^* \to 0$ and $V \to 0$ as $t \to \infty$. Estimate this value.

2. Show that if $\eta < \eta_{crit}$, then that T, T^*, and V tend to constants as $t \to \infty$; say $T \to T_{\infty,\eta}$, $T^* \to T_{\infty,\eta}^*$, and $V \to V_{\infty,\eta}$.

3. Doctors say AIDS sets in if $T < 200$. Show that there is a critical value η_{AIDS} so that if $\eta > \eta_{AIDS}$, then $T_{\infty,\eta}^* \geq 200$. Use you simulation to estimate this value.

4. Find analytically the value of η_{AIDS}. Compare it to the results of your simulation.

Write up a technical report that answers these questions. The report should describe the model, the numerical methods used to solve the problem, your program, and your results. When answering these questions, you must address the question of how the choice of step size affects the result.

The program should be written using good object oriented programming techniques.

Acknowledgements

This project was inspired by the excellent paper of Perelson & Nelson, *Mathematical Analysis of HIV-1 Dynamics in Vivo*, SIAM Review **41**(1) pp. 3-44, 1999. The values of the parameters in the project were taken from Perelson, Kirschner and De Boer, *Dynamics of HIV Infection of CD4+ T cells*, Math. Biosci., **114** pp. 81-125, 1993.

Project: The Double Pendulum

Section 1: Introduction

In this project, we will construct a model for the double pendulum. A double pendulum is a pendulum attached to the end of a second pendulum, the whole thing being constrained to move in a single plane.

The model for a single pendulum is relatively easy to derive; however the double pendulum is much more complex. The best approach to modeling the double pendulum is to use Lagrangian dynamics, which was shall introduce.

For the project, we will determine if the system has sensitive dependence on initial conditions. In particular, do small changes in the initial positions of the pendulum cause large variations in the subsequent motion of the double pendulum?

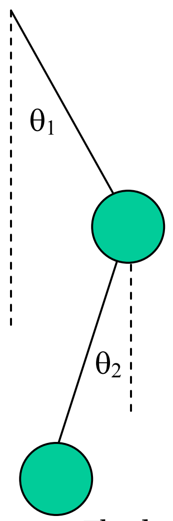


Figure 1: The double pendulum

Section 2: Modeling the Single Pendulum

Let us begin with a review of the basic physics of rotating bodies.

To find the *torque* τ exerted by a force on a body about a given axis, first find the lever arm, which is the line segment from the axis to the point at which the force acts. Torque is the product of the length of the lever arm with the component of the force in the direction perpendicular to the lever arm.

The *moment of inertia* I of a collection of point masses m_i about an axis is

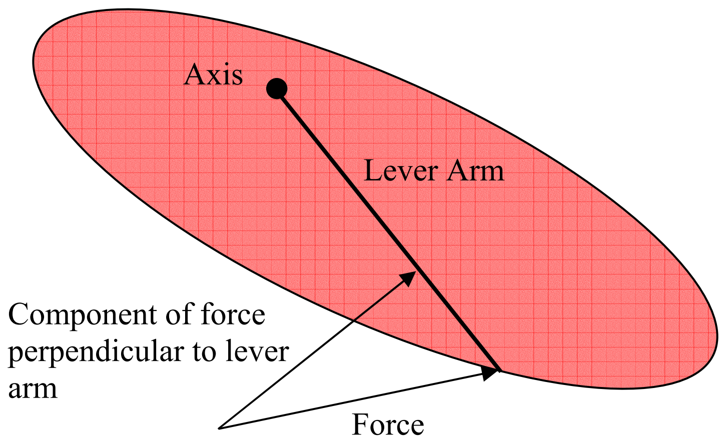


Figure 2: Torque

the sum $\sum m_i r_i$ where r_i is the distance from mass i to the axis. In a continuous body, the sum is replaced by an appropriate integral.

Newton's Law for angular motion then says that

$$\tau = I\alpha$$

where α is the angular acceleration about the axis.

Consider the pendulum with mass m and arm of length L shown in Figure 3. To determine its motion, let $\theta = \theta(t)$ be the angle the pendulum makes with the vertical. The torque on the pendulum is the product of the component of the force perpendicular to the arm multiplied by the length of the arm. The only

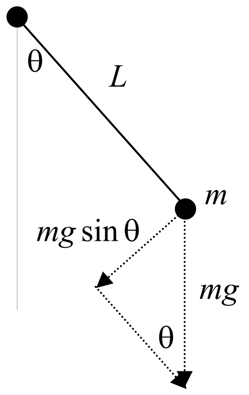


Figure 3: The single pendulum

force acting on the pendulum is that of gravity. If we assume for simplicity that the mass of the arm is negligible relative to the mass at the end of the arm, then the force of gravity is simply mg pointing directly downward. The component of this force perpendicular to the arm is then $-mg\sin\theta$, and hence the torque is

$$\tau = -Lmg\sin\theta .$$

The negative sign must be included because the torque due to gravity acts in the direction opposite that of the angle θ.

The moment of inertia of the single pendulum is the product of the mass and the length of the arm m; hence $I = Lm$. Finally, because the angle the pendulum makes with the vertical is θ, we know that

$$\alpha = \theta'' .$$

Combining these with Newton's Law, we find that

$$\theta'' = -\frac{g}{L}\sin\theta . \qquad (1)$$

Section 3: Energy Methods

There is another way that we can derive the model (1). The *kinetic energy* of an object of mass m moving at velocity v is the quantity $\frac{1}{2}mv^2$. On the other hand, the gravitational potential energy of an object of mass m and height h is mgh. The principle of conservation of energy says that energy is conserved; in particular it says that for our pendulum the sum of the kinetic energy and the potential energy should be constant.

For the pendulum, the gravitational potential energy is easy to calculate; trigonometry shows us that the height of the mass is $-L\cos\theta$ when measured

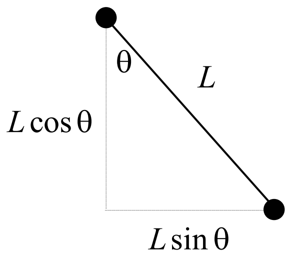


Figure 4: The single pendulum

from the axis of the pendulum. Thus

$$PE = -mgL\cos\theta.$$

On the other hand, to find the kinetic energy of the pendulum, we begin by noting that the coordinates of the mass of the pendulum are $x = L\sin\theta$, $y = -L\cos\theta$ measured from the axis of the pendulum. Then, if v is the velocity of the mass of the pendulum, we have

$$\begin{aligned} v^2 &= (x')^2 + (y')^2 \\ &= L^2\sin^2\theta\cdot(\theta')^2 + L^2\cos^2\theta\cdot(\theta')^2 \\ &= L^2(\theta')^2. \end{aligned}$$

Thus

$$KE = \tfrac{1}{2}mL^2(\theta')^2.$$

Conservation of energy then requires that

$$KE + PE = \tfrac{1}{2}mL^2(\theta')^2 - mgL\cos\theta$$

is a constant. This means that its derivative is necessarily zero, and thus

$$\left(\tfrac{1}{2}mL^2\right)\cdot 2\theta'\theta'' + mgL\sin\theta\cdot\theta' = 0.$$

Solving for θ'', we obtain (1) again.

Section 4: Hamilton's Principle

Although both of the previous methods suffice to construct a model for the single pendulum, the double pendulum is more complex. To model it, we shall introduce Hamilton's principle and Lagrangian dynamics.

Suppose that we have physical system that can be determined by the values of n parameters $q_1, q_2, \ldots, q_n$ (called *degrees of freedom*). Suppose also that the mechanical properties of the system are determined by two quantities, the potential energy and the kinetic energy. Assume that the kinetic energy KE depends on the positions q_i, the velocities q_i', and the time t; suppose also that KE has the form

$$KE = \sum_{i,j=1}^{n} K_{ij}\left(q_1, q_2, \ldots, q_n; t\right) q_i' q_j'$$

so that it is quadratic in the velocities q_i'. We also assume that the potential energy PE is a function of the positions q_i and the time t, and that it does not depend on the velocities q_i'. Hamilton's principle then says that between any two instants of time t_0 and t_1 the system chooses functions $q_i(t)$ so that the integral

$$J\left[q_1, q_2, \ldots, q_n\right] = \int_{t_0}^{t_1} \left(KE - PE\right)\ dt$$

is stationary .

What do we mean when we say that J is stationary? Suppose that the initial state $q_i\left(t_0\right)$ and the final state $q_i\left(t_1\right)$ are given. Let $p_i\left(t\right)$ be functions so that $p_i\left(t_0\right) = p_i\left(t_1\right) = 0$. Then for any real number λ, the functions $q_i\left(t\right) + \lambda p_i\left(t\right)$ have the same initial and final states. Now consider λ as the variable, and examine the function

$$F\left(\lambda\right) = J\left[q_1 + \lambda p_1, q_2 + \lambda p_2, \ldots, q_n + \lambda p_n\right].$$

If $F\left(\lambda\right)$ has a minimum when $\lambda = 0$ no matter what choice of p_i are made, then the original choices of q_i make J a minimum. Further, because $F\left(\lambda\right)$ has a minimum when $\lambda = 0$, we know that $F'\left(0\right) = 0$. In general, Hamilton's principle attempts to minimize the value of J. However, this does not always occur; what does occur is that, even though $F\left(\lambda\right)$ might not have a minimum at $\lambda = 0$, we must have $F'\left(0\right) = 0$. In this case, we say that the integral is *stationary*.

The quantity $\mathsf{L} = KE - PE$ has a special name; it is called the *Lagrangian* of the motion.

To better understand Hamilton's principle, let us apply it to the single pendulum. In this case the state of the system is determined by just one parameter, θ, the angle the pendulum makes with the vertical. Thus this problem has just one degree of freedom.

From Section 3, we see that the kinetic energy is

$$KE = \tfrac{1}{2} mL^2 \left(\theta'\right)^2$$

and the potential energy is

$$PE = -mgL\cos\theta,$$

thus the Lagrangian for the single pendulum is

$$\mathsf{L} = \frac{1}{2} mL^2 (\theta')^2 + mgL\cos\theta .$$

Hamilton's principle says that the motion will proceed so that the integral

$$J[\theta] = \int_{t_0}^{t_1} \left\{ \frac{1}{2} mL^2 (\theta')^2 + mgL\cos\theta \right\} dt$$

is stationary. To find the stationary points, let $\phi(t)$ be any function with $\phi(t_0) = \phi(t_1) = 0$, and let λ be a real number. Define

$$F(\lambda) = J[\theta + \lambda\phi] = \int_{t_0}^{t_1} \left\{ \frac{1}{2} mL^2 (\theta' + \lambda\phi')^2 + mgL\cos(\theta + \lambda\phi) \right\} dt .$$

We can then calculate

$$F'(\lambda) = \int_{t_0}^{t_1} \left\{ mL^2 (\theta' + \lambda\phi')\phi' - mgL\sin(\theta + \lambda\phi)\cdot\phi \right\} dt .$$

Thus

$$F'(0) = \int_{t_0}^{t_1} \left\{ mL^2\theta'\phi' - mgL(\sin\theta)\cdot\phi \right\} dt .$$

Integrate by parts in the first term and use the fact that $\phi(t_0) = \phi(t_1) = 0$ to see that

$$F'(0) = \int_{t_0}^{t_1} \left\{ -mL^2\theta'' - mgL\sin\theta \right\} \phi \quad dt = 0$$

for every ϕ. Knowing that an integral is zero does not tell us much about the integrand, however here we are presented with a special case. We know that

$$\int_{t_0}^{t_1} \left\{ -mL^2\theta'' - mgL\sin\theta \right\} \phi \quad dt = 0$$

for *every* choice of ϕ with $\phi(0) = \phi(1) = 0$. The only way that this can happen for every choice of ϕ is if the term in the brackets is identically zero; this means that

$$-mL^2\theta'' - mgL\sin\theta = 0$$

which simplifies to

$$\theta'' = -\frac{g}{L}\sin\theta$$

which is exactly (1) once again.

Section 5: Lagrange's Equations of Motion

Hamilton's principle is very powerful, but it is also quite cumbersome. It would be nice if the process could be streamlined. In fact, it can, but it requires some knowledge of partial derivatives.

Partial Derivatives

Consider a function of two variables $f(x,y)$. If we hold the variable y fixed, the result is a function of one variable $x \mapsto f(x,y)$. The derivative of this function is the *partial derivative of* $f(x,y)$ *with respect to* x. It is written as f_x

or as $\frac{\partial f}{\partial x}$. Similarly, if we hold x fixed, we obtain the partial derivative of $f(x,y)$ with respect to y, written f_y or as $\frac{\partial f}{\partial y}$. In general we have the definition

$$\frac{\partial f}{\partial x}(x,y)=\lim_{h\to 0}\frac{f(x+h,y)-f(x,y)}{h}$$

with a similar expression for $\frac{\partial f}{\partial y}$.

We shall not discuss partial derivatives in detail, but we do need the following Chain Rule. If $x=x(t)$ and $y=y(t)$, then

$$\frac{d}{dt}f(x(t),y(t))=\frac{\partial f}{\partial x}\frac{dx}{dt}+\frac{\partial f}{\partial y}\frac{dy}{dt}.$$

To see why, note the following argument.

$$\begin{aligned}\frac{d}{dt}f(x(t),y(t))&=\lim_{h\to 0}\frac{f(x(t+h),y(t+h))-f(x(t),y(t))}{h}\\&=\lim_{h\to 0}\left\{\frac{f(x(t+h),y(t+h))-f(x(t),y(t+h))}{h}+\frac{f(x(t),y(t+h))-f(x(t),y(t))}{h}\right\}\\&=\lim_{h\to 0}\frac{f(x(t+h),y(t+h))-f(x(t),y(t+h))}{x(t+h)-x(t)}\lim_{h\to 0}\frac{x(t+h)-x(t)}{h}\\&\qquad+\lim_{h\to 0}\frac{f(x(t),y(t+h))-f(x(t),y(t))}{y(t+h)-y(t)}\lim_{h\to 0}\frac{y(t+h)-y(t)}{h}\\&=\frac{\partial f}{\partial x}\frac{dx}{dt}+\frac{\partial f}{\partial y}\frac{dy}{dt}\end{aligned}$$

Although this argument is not sufficient to be a proof (Why?), it explains the basic idea.

Applications to Hamilton's Principle

Let $KE=KE(q_1',q_2',\ldots,q_n',q_1,q_2,\ldots,q_n;t)$ be the kinetic energy of a system, and let $PE=PE(q_1,q_2,\ldots,q_n;t)$ be its potential energy. The Lagrangian is the function

$$\mathcal{L}(q',q;t)=KE(q',q;t)-PE(q,t).$$

Hamilton's principle tells us that the motion proceeds so that the integral $J=\int_{t_0}^{t_1}\mathcal{L}\,dt$ is stationary. Let $\phi=(\phi_1,\phi_2,\ldots,\phi_n)$be any function with $\phi_i(t_0)=\phi_i(t_1)=0$ for each i, and consider

$$F(\lambda)=J[q_1+\lambda p_1,q_2+\lambda p_2,\ldots,q_n+\lambda p_n].$$

We know that $F'(\lambda)=0$ when $\lambda=0$ Using partial derivatives, we can evaluate the derivative $F'(\lambda)$. Indeed

$$\frac{d}{d\lambda}F(\lambda) = \frac{d}{d\lambda}\int_{t_0}^{t_1} \mathsf{L}\left(q' + \lambda\phi', q + \lambda\phi; t\right)\ dt$$
$$= \sum_{i=1}^{n}\int_{t_0}^{t_1}\left\{\frac{\partial \mathsf{L}\left(q' + \lambda\phi', q + \lambda\phi; t\right)}{\partial \dot{q}_i}\phi_i' + \frac{\partial \mathsf{L}\left(q' + \lambda\phi', q + \lambda\phi; t\right)}{\partial q_i}\phi_i\right\}dt$$

Thus, if we integrate by parts in the first term, we see that

$$F(0) = \sum_{i=1}^{n}\int_{t_0}^{t_1}\left\{-\frac{d}{dt}\frac{\partial \mathsf{L}\left(q', q; t\right)}{\partial q_i'} + \frac{\partial \mathsf{L}\left(q', q; t\right)}{\partial q_i}\right\}\phi_i'\ dt$$

Then, because the functions ϕ_1, ϕ_2, ..., ϕ_n can be chosen freely, we conclude that the integrand is zero, and hence

$$\frac{\partial \mathsf{L}}{\partial q_i} - \frac{d}{dt}\frac{\partial \mathsf{L}}{\partial q_i'} = 0 \qquad (2)$$

for each $i = 1, 2, \ldots, n$. These equations are called *Lagrange's Equations of Motion*. Though they use partial derivatives, they are much simpler to apply than the cumbersome process of finding the stationary points of

$J\left[q_1, q_2, \ldots, q_n\right] = \int_{t_0}^{t_1}\left(KE - PE\right)\ dt$.

Applications to the Single Pendulum

We saw in the previous section that, for the single pendulum

$$\mathsf{L} = \frac{1}{2}mL^2\left(\theta'\right)^2 + mgL\cos\theta .$$

In this case, there is only one variable, so that $n = 1$ and $q = \theta$. Then

$$\frac{\partial \mathsf{L}}{\partial q} = \frac{\partial \mathsf{L}}{\partial \theta} = -mgL\sin\theta,$$
$$\frac{\partial \mathsf{L}}{\partial q'} = \frac{\partial \mathsf{L}}{\partial \theta'} = mL^2\theta'.$$

Thus equation (2) becomes

$$\theta'' = -\frac{g}{L}\sin\theta .$$

Section 6: The Double Pendulum

Now we will use Lagrange's Equations of Motion to construct a model for the double pendulum. Suppose that the first pendulum has a bob with mass m_1 and arm of length L_1. To this we attach a second pendulum whose bob has mass m_2 with an arm of length L_2.

We begin by calculating the potential energy of the system. Referring to Figure 5, we see that the height of mass 1 is $-L_1 \cos\theta_1$, while the height of the second mass is

$$-L_1 \cos\theta_1 - L_2 \cos\theta_2 .$$

Thus, the potential energy is

$$PE = -m_1 g L_1 \cos\theta_1 - m_2 g\left(L_1 \cos\theta_1 + L_2 \cos\theta_2\right).$$

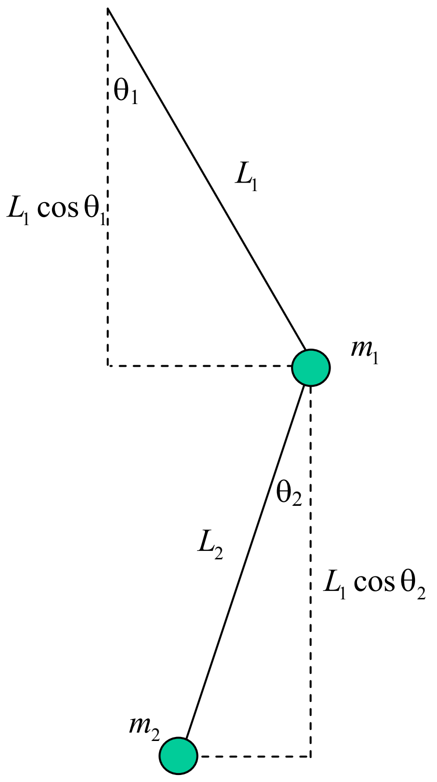


Figure 5: The double pendulum

Next, we find the kinetic energy of the system. Mass m_1 is at position $x_1 = L_1 \sin\theta_1$, $y_1 = -L_1 \cos\theta_1$ so that

$$x_1' = L_1\theta_1' \cos\theta_1$$
$$y_1' = L_1\theta_1' \sin\theta$$

Its velocity v_1 satisfies

$$\left(v_1\right)^2 = \left(x_1'\right)^2 + \left(y_1'\right)^2$$

so that it has kinetic energy

$$KE_1 = \tfrac{1}{2}m_1\left[\left(L_1\theta_1' \cos\theta_1\right)^2 + \left(L_1\theta_1' \sin\theta\right)^2\right]$$
$$= \tfrac{1}{2}m_1 L_1^2\left(\theta_1'\right)^2 .$$

On the other hand, mass m_2 is at position $x_2 = L_1 \sin\theta_1 + L_2 \sin\theta_2$, $y_2 = -L_1 \cos\theta_1 - L_2 \cos\theta_2$. Thus

$$x_2' = L_1\theta_1' \cos\theta_1 + L_2\theta_2' \cos\theta_2,$$
$$y_2' = L_1\theta_1' \sin\theta + L_2\theta_2' \sin\theta_2.$$

It has velocity v_2, where

$$\begin{aligned} v_2^2 &= \left(x_2'\right)^2 + \left(y_2'\right)^2 = \left(L_1\theta_1' \cos\theta_1 + L_2\theta_2' \cos\theta_2\right)^2 + \left(L_1\theta_1' \sin\theta + L_2\theta_2' \sin\theta_2\right)^2 \\ &= \left(L_1\theta_1' \cos\theta_1\right)^2 + \left(L_1\theta_1' \sin\theta\right)^2 + \left(L_2\theta_2' \cos\theta_2\right)^2 + \left(L_2\theta_2' \sin\theta_2\right)^2 \\ &\qquad + 2L_1L_2\theta_1'\theta_2'\left(\cos\theta_1 \cos\theta_2 + \sin\theta_1 \sin\theta_2\right) \\ &= L_1^2\left(\theta_1'\right)^2 + L_2^2\left(\theta_2'\right)^2 + 2L_1L_2\theta_1'\theta_2' \cos\left(\theta_2 - \theta_1\right). \end{aligned}$$

Thus it has kinetic energy

$$KE_2 = \tfrac{1}{2}m_2\left[L_1^2\left(\theta_1'\right)^2 + L_2^2\left(\theta_2'\right)^2 + 2L_1L_2\theta_1'\theta_2' \cos\left(\theta_2 - \theta_1\right)\right].$$

Combining these, we see that the kinetic energy of our complete system is

$$KE = \tfrac{1}{2}\left(m_1 + m_2\right)L_1^2\left(\theta_1'\right)^2 + \tfrac{1}{2}m_2L_2^2\left(\theta_2'\right)^2 + m_2L_1L_2\theta_1'\theta_2' \cos\left(\theta_2 - \theta_1\right).$$

The Lagrangian L is given by

$$\begin{aligned} \mathsf{L} &= \tfrac{1}{2}\left(m_1 + m_2\right)L_1^2\left(\theta_1'\right)^2 + \tfrac{1}{2}m_2L_2^2\left(\theta_2'\right)^2 + m_2L_1L_2\theta_1'\theta_2' \cos\left(\theta_2 - \theta_1\right) \\ &\qquad + m_1 g L_1 \cos\theta_1 + m_2 g\left(L_1 \cos\theta_1 + L_2 \cos\theta_2\right) \end{aligned}$$

We have two degrees of freedom, with $q_1 = \theta_1$ and $q_2 = \theta_2$. Then

$$\frac{\partial \mathsf{L}}{\partial q_1} = \frac{\partial \mathsf{L}}{\partial \theta_1} = m_2 L_1 L_2 \theta_1' \theta_2' \sin(\theta_2 - \theta_1) - m_1 g L_1 \sin\theta_1 - m_2 g L_1 \sin\theta_1$$
$$= -(m_1 + m_2) g L_1 \sin\theta_1 + m_2 L_1 L_2 \theta_1' \theta_2' \sin(\theta_2 - \theta_1)$$

while

$$\frac{\partial \mathsf{L}}{\partial q_1'} = \frac{\partial \mathsf{L}}{\partial \theta_1'} = (m_1 + m_2) L_1^2 \theta_1' + m_2 L_1 L_2 \theta_2' \cos(\theta_2 - \theta_1).$$

Applying (2), we see that

$$-(m_1 + m_2) g L_1 \sin\theta_1 + m_2 L_1 L_2 \theta_1' \theta_2' \sin(\theta_2 - \theta_1)$$
$$= \frac{d}{dt}\left[(m_1 + m_2) L_1^2 \theta_1' + m_2 L_1 L_2 \theta_2' \cos(\theta_2 - \theta_1)\right].$$

Thus

$$-(m_1 + m_2) g L_1 \sin\theta_1 + m_2 L_1 L_2 \theta_1' \theta_2' \sin(\theta_2 - \theta_1)$$
$$= \left[(m_1 + m_2) L_1^2 \theta_1'' + m_2 L_1 L_2 \theta_2'' \cos(\theta_2 - \theta_1) - m_2 L_1 L_2 \theta_2' (\theta_2' - \theta_1') \sin(\theta_2 - \theta_1)\right]$$

and hence

$$(m_1 + m_2) L_1^2 \theta_1'' + m_2 L_1 L_2 \cos(\theta_2 - \theta_1) \cdot \theta_2''$$
$$= -(m_1 + m_2) g L_1 \sin\theta_1 + m_2 L_1 L_2 (\theta_2')^2 \sin(\theta_2 - \theta_1). \qquad (3)$$

Next we calculate

$$\frac{\partial \mathsf{L}}{\partial \theta_2} = -m_2 L_1 L_2 \theta_1' \theta_2' \sin(\theta_2 - \theta_1) - m_2 g L_2 \sin\theta_2$$

and

$$\frac{\partial \mathsf{L}}{\partial \theta_2'} = m_2 L_2^2 \theta_2' + m_2 L_1 L_2 \theta_1' \cos(\theta_2 - \theta_1).$$

Applying (2) once again, we find that

$$-m_2 L_1 L_2 \theta_1' \theta_2' \sin(\theta_2 - \theta_1) - m_2 g L_2 \sin\theta_2 = \frac{d}{dt}\left[m_2 L_2^2 \theta_2' + m_2 L_1 L_2 \theta_1' \cos(\theta_2 - \theta_1)\right].$$

Thus

$$-m_2 L_1 L_2 \theta_1' \theta_2' \sin(\theta_2 - \theta_1) - m_2 g L_2 \sin\theta_2$$
$$= m_2 L_2^2 \theta_2'' + m_2 L_1 L_2 \theta_1'' \cos(\theta_2 - \theta_1) - m_2 L_1 L_2 \theta_1' (\theta_2' - \theta_1') \sin(\theta_2 - \theta_1)$$

and hence

$$m_2 L_1 L_2 \cos(\theta_2 - \theta_1) \cdot \theta_1'' + m_2 L_2^2 \theta_2''$$
$$= -m_2 g L_2 \sin\theta_2 - m_2 L_1 L_2 (\theta_1')^2 \sin(\theta_2 - \theta_1). \qquad (4)$$

Thus, if combine (3) and (4), we see that the system of equations we are to solve is

$$\begin{cases} (m_1 + m_2) L_1 \theta_1'' + m_2 L_2 \cos(\theta_2 - \theta_1) \cdot \theta_2'' = -(m_1 + m_2) g \sin\theta_1 + m_2 L_2 (\theta_2')^2 \sin(\theta_2 - \theta_1) \\ L_1 \cos(\theta_2 - \theta_1) \cdot \theta_1'' + L_2 \theta_2'' = -g \sin\theta_2 - L_1 (\theta_1')^2 \sin(\theta_2 - \theta_1). \end{cases}$$

To this system we add appropriate initial conditions; these are the values of $\theta_1(0)$, $\theta_2(0)$, $\theta_1'(0)$, and $\theta_2'(0)$ which are the positions and angular velocities when $t = 0$.

Assignments

1. Use Taylor's theorem to prove that, for small values of θ that $\sin\theta \approx \theta$. Estimate the error. Apply this to the single pendulum and explain why, for small angles θ, that $\theta'' = -\frac{g}{L}\theta$.

2. By looking at trigonometric functions, find at least one solution of the equation $\theta'' = -\frac{g}{L}\theta$.

3. By looking at trigonometric functions, find the function that satisfies

$$\begin{cases} \theta'' = -\frac{g}{L}\theta \\ \theta(0) = \theta_0 \\ \theta'(0) = \omega_0 \end{cases}$$

What is the period of these solutions? What does that say about the period of the pendulum?

We *know* that the shortest distance between two points is a straight line. In questions 4-9 we shall prove this fact.

4. Let $y(t)$ be a function for which $y(0) = a$ and $y(1) = b$. Write down an integral that gives the length of this curve.

5. Explain why the choice of function $y(t)$ which has the shortest length is stationary for the integral

$$J[y] = \int_0^1 \sqrt{1+(y')^2}\,dt.$$

6. To find the stationary points, let $\phi(t)$ be any function with $\phi(0) = \phi(1) = 0$, let λ be a real number, and set

$$F(\lambda) = J[y + \lambda\phi].$$

Explain why we know that if $y(t)$ has the shortest length then $F'(0) = 0$.

7. Show that $F'(0) = \int_0^1 \frac{y'\phi'}{\sqrt{1+(y')^2}}\,dt$.

8. Explain why we know that if $y(t)$ has the shortest length then

$$\frac{d}{dt}\left(\frac{y'}{\sqrt{1+(y')^2}}\right) = 0.$$

9. Solve this equation to conclude that y' is constant, and that consequently $y(t)$ is a straight line.

10. What is the flaw in the argument used to show why $\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$?

Project

Write a C++ program that simulates the motion of a double pendulum.

As input, the program should take

- The initial positions and angular velocities of the two bobs,
- The step size for the simulation, and
- The total time for the simulation.

As output, the program should return

- Graphs of angle versus time for each of the angles in the double pendulum,
- A graphical representation of the motion of the bodies, and
- The final positions and angular velocities of the two bobs.

The program should let the user set the position of the double pendulum using the mouse or a text input. Changes in the initial position of the pendulum should be reflected in the graph of the double pendulum, before the simulation begins. The user should be able to choose the scale in each of the angle versus time graphs. The simulation should use a fourth order Runge-Kutta scheme.

The program should be written using good object oriented programming techniques.

You are to use your simulation to investigate the stability of the double pendulum as the initial data is modified.

You are then to write up a technical report that answers the following questions:

- What is the mathematical model of the problem?
- What is the numerical method used to solve the problem?
- What is the structure of your program?

The differential equation

$$\begin{cases} y' = f(t, y) \\ y(t_0) = y_0 \end{cases}$$

displays *sensitive dependence on initial conditions* at (t_0, y_0) if small changes in either t_0 or y_0 can produce significant changes in the solution.

Does the double pendulum display sensitive dependence on initial conditions? If so, does this always happen, or is it true only for some data?

When answering these questions, it is *essential* that you address the question of how the choice of step size affects the result. You *must* run the simulation for different step sizes to draw correct conclusions.

The Mouse

Section 1: Introduction

Many programs use the mouse to perform essential functions. To illustrate how we can integrate the mouse into our code, we shall write a simple program that uses the mouse to drag a ball across the screen.

Section 2: The Skeleton

Create a dialog based program called Mouse. The main dialog requires no functionality save for the OK button which we rename Exit Program. We add a new dialog called `IDD_GRAPH` where the drawing will take place. We remove the controls placed there by the default creation process, and uncheck the System Menu box from the Styles property tab from the Properties dialog box.

We use the Class Wizard to associate this dialog box with the class `CGraph`. We add just one private variable to the class, of type `CPoint` called `m_ptCenter`. This will hold the value of the center of the ball we plan to draw on the screen. To initialize this variable correctly, we add a message handler for `WM_INITDIALOG` and add the code

```
BOOL CGraph::OnInitDialog()
{
      CDialog::OnInitDialog();

      // TODO: Add extra initialization here

      CRect MainWindow;
      GetClientRect(MainWindow);
      m_ptCenter = MainWindow.CenterPoint();

      return TRUE;  // return TRUE unless you set the focus to a
control
                    // EXCEPTION: OCX Property Pages should
return FALSE
}
```

To display the window, we add a private variable `m_wndGraph` to the `CMouseDlg` class, and add the following code to the `OnInitDialog()` method of the `CMouseDlg` class:

```
      m_wndGraph.Create(IDD_GRAPH);
      m_wndGraph.ShowWindow(SW_SHOW);
```

Finally, we add a message handler for the `WM_PAINT` message for `CGraph` and add the following code

```
void CGraph::OnPaint()
{
      CPaintDC dc(this); // device context for painting

      // TODO: Add your message handler code here

      CBrush* OldBrush;
      CBrush GreenBrush;

      GreenBrush.CreateSolidBrush(RGB(0,255,0));
      OldBrush = dc.SelectObject(&GreenBrush);

      dc.Ellipse(m_ptCenter.x-25,m_ptCenter.y-25,
            m_ptCenter.x+25,m_ptCenter.y+25);

      dc.SelectObject(OldBrush);

      // Do not call CDialog::OnPaint() for painting messages
}
```

If we compile and run this program, it will simply display a green ball in the center of the screen.

Section 3: The Mouse Handler

To use the mouse in our code, we must capture and use the messages that windows sends to our program that describe the state of the mouse. There are a number of different possible windows messages that can be sent when the user uses the mouse. These include:

- `WM_MOUSEMOVE`
- `WM_LBUTTONDOWN`
- `WM_LBUTTONUP`
- `WM_LBUTTONDBLCLK`
- `WM_RBUTTONDOWN`
- `WM_RBUTTONUP`
- `WM_RBUTTONDBLCLK`

To see how these windows messages can be used, we shall write a handler the WM_MOUSEMOVE method and use it to move the ball around the screen. The other messages can be handled in a similar fashion, and used to extend the functionality of our program.

From the CGraph class, add a message handler for WM_MOUSEMOVE. You will be presented with the following default code

```
void CGraph::OnMouseMove(UINT nFlags, CPoint point)
{
      // TODO: Add your message handler code here and/or call
         default
```

```
        CDialog::OnMouseMove(nFlags, point);
}
```

The `CPoint` variable point contains the screen coordinates of the mouse. The `nFlags` variable is an integer that contains flags that describe which keys and mouse buttons are down. Options for this variable include

- `MK_CONTROL` Set if the Control key is down.
- `MK_LBUTTON` Set if the left mouse button is down.
- `MK_RBUTTON` Set if the right mouse button is down.
- `MK_SHIFT` Set if the Shift key is down.

Because these are used as flags however, some subtlety is required in their use. If we want to check if the Shift key is down, we can not use code like the following

```
if(nFlags == MK_SHIFT)
{
        // Do something cool
}
```

This is because the value for the `nFlags` variable is a bitwise combination of all of the possible combinations. Thus, if the Shift key and the left mouse button were both pressed, we would not have `nFlags=MK_SHIFT`, nor would we have `nFlags=MK_LBUTTON`. Instead we would have

```
nFlags= MK_SHIFT | MK_LBUTTON
```

provided no other keys / mouse buttons were pressed. Recall here that | is bitwise OR.

Thus, we modify the `OnMouseMove` method to read as follows

```
void CGraph::OnMouseMove(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call
        default

        CDialog::OnMouseMove(nFlags, point);

        if((nFlags & MK_LBUTTON) == MK_LBUTTON )
        {
                m_ptCenter = point;
                Invalidate();
        }
}
```

Note how we used the bitwise AND operator & to determine if the left mouse button was pressed.

If we compile and run this code, we will be able to use the mouse to drag our ball to any point on the screen.

Section 4: Resizing the Window

All the dialog boxes we have used so far have been of a fixed width. However, it is a simple matter to allow the user to modify the size of a dialog box as the code is running. Indeed, from the Properties menu for the dialog box, select the Styles tab, and change the setting in the Border pull-down box to Resizing.

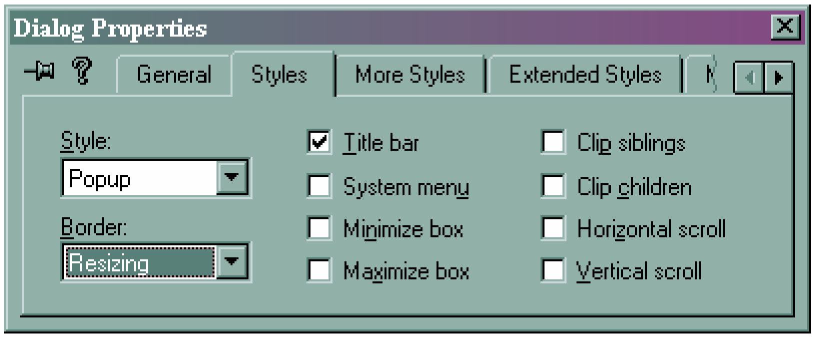


Figure 1: Allowing a dialog box to be resized

Assignments

1. What would the program do if you used the mouse and clicked just one single point? Explain your answer by referring to the code. Modify the program so that it behaves more intuitively.
2. Modify this program so that the clicking the right mouse button is used to modify the size of the ellipse.

Project: Diffusion

Section 1: Introduction

Suppose that we have an industrial plant emitting smog into the atmosphere at some rate. We would like to know the concentration of smog at different distances from the plant.

In this project, we shall construct a model for a diffusion process; this model will be in the form of a partial differential equation. We will learn how to solve this equation numerically using finite difference methods. We shall briefly discuss the convergence, consistency, and stability properties of finite difference methods.

Section 2: The Model

Method #1: System of Differential Equations

Suppose that the plant is located at $x = 0$. Consider the interval $[0, L]$ for some real number L. Subdivide this region into n subintervals, and let $h = L/n$ be the width of one subinterval. Let $x_0 = 0$, $x_1 = h$, $x_2 = 2h$, ..., $x_{n-1} = (n-1)h$, and $x_n = nh = L$ be the endpoints of the subintervals. Let u be the concentration of smog in the air; in particular let $u_i(t)$ be the concentration of smog at the point x_i at time t.

Suppose that rate of change of the smog concentration at a point is proportional to the difference between the concentrations at adjacent points. Our model becomes

$$u_i'(t) = c\left[u_{i+1} - 2u_i + u_{i-1}\right] \tag{1}$$

for some constant of proportionality c, at least for $1 \le i \le n-1$. Note that this equation can not hold when $i = 0$ or $i = n$ because these are the endpoints. Thus, we have $n-1$ equations in the $n+1$ variables $u_i(t)$, and we need to add boundary conditions to solve this problem. Provided the boundary conditions are specified, this system can be solved by using, for example, a Runge-Kutta method.

The form of the right hand side of (1) has special significance. Let $f(x)$ be a function. Then Taylor's theorem tells us that

$$f(x+h) = f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(x) + \tfrac{1}{6}h^3 f'''(x) + \tfrac{1}{24}h^4 f^{(4)}(x) + O\left(h^5\right)$$

and

$$f(x-h) = f(x) - hf'(x) + \tfrac{1}{2}h^2 f''(x) - \tfrac{1}{6}h^3 f'''(x) + \tfrac{1}{24}h^4 f^{(4)}(x) + O\left(h^5\right)$$

so that if we add these equations, we see that

$$f(x+h)+f(x-h)=2f(x)+h^2f''(x)+O(h^4).$$

Hence

$$f(x+h)-2f(x)+f(x-h)=h^2f''(x)+O(h^4)$$

As a consequence, if $u=u(x,t)$, our model can be thought of as an approximation to the equation $u_t=(h^2c)u_{xx}=\gamma u_{xx}$.

Partial Derivatives

Consider a function of two variables $f(x,y)$. If we hold the variable y fixed, the result is a function of one variable $x\mapsto f(x,y)$. The derivative of this function is the *partial derivative of $f(x,y)$ with respect to x*. It is written as f_x or as $\frac{\partial f}{\partial x}$. Similarly, if we hold x fixed, we obtain the partial derivative of $f(x,y)$ with respect to y, written f_y or as $\frac{\partial f}{\partial y}$. In general

$$\frac{\partial f}{\partial x}(x,y)=\lim_{h\to 0}\frac{f(x+h,y)-f(x,y)}{h},$$

with a similar expression for $\frac{\partial f}{\partial y}$.

Model #2: Partial Differential Equation

Let $u=u(x,t)$ be the concentration of smog at the point x at time t. Choose any interval $(a,b)\subset[0,L]$. The total amount of smog in (a,b) is $\int_a^b u(x,t)\,dx$, so the rate of change is

$$\frac{d}{dt}\int_a^b u(x,t)\,dx=\int_a^b u_t(x,t)\,dx.$$

Where did the smog go? It must have left the interval through the endpoints. If the smog concentration on both sides of a point is the same, then we expect that there is no diffusion of smog across that point. On the other hand, if there is a significant difference in the smog concentration, we expect the smog to quickly diffuse from the area of higher concentration to the area of lower concentration. Thus, we expect that the rate smog diffuses across a point depends on the derivative u_x. For simplicity, we suppose it leaves the endpoints at the rate γu_x. Thus smog leaves the interval (a,b) at the rate

$$\gamma u_x\big|_a^b=\int_a^b(\gamma u_x(x,t))_x\,dx.$$

Combining these, we find that, for any interval (a,b) that

$$\int_a^b(\gamma u_x(x,t))_x\,dx=\int_a^b u_t(x,t)dx,$$

and hence

$$\int_a^b \left[u_t(x,t) - \gamma u_{xx} \right] \quad dx = 0 .$$

Because the interval (a,b) is arbitrary, we conclude that

$$u_t = \gamma u_{xx} .$$

Model #3: Probabilistic Model

Let u_i^j be the concentration of particles at the point $x_i = ih$ at time $t_n = jk$. Here k is a chosen unit of time. Let p be the probability that a smog particle at x_i will jump to x_{i+1} in time k; we also let p be the probability that a smog particle at x_i will jump to x_{i-1} in the same time period. The probability that it remains at x_i is then $1-2p$. Then

$$u_i^{j+1} = pu_{i-1}^j + pu_{i+1}^j + (1-2p)u_i^j .$$

This is called the *Chapman-Kolmogorov* equation of the process. Rewrite this equation as

$$\frac{u_i^{j+1} - u_i^j}{k} = \left(p\frac{h^2}{k} \right) \frac{u_{i+1}^j - 2u_i^j + u_{i-1}^j}{h^2} .$$

As $h,k \to 0$, this equation becomes $u_t = \gamma u_{xx}$, provided $\gamma = ph^2/k$. (The limiting process here is very delicate!) This last equation

$$u_t = \gamma u_{xx} ,$$

is called the *Fokker-Planck* equation for the continuous process.

Generalizations

The preceding derivations apply to problems more general than to smog. They apply more generally to diffusion processes. As an example, these derivations also apply to the flow of heat. The equation $u_t = \gamma u_{xx}$ is called the heat equation, or the diffusion equation.

Section 3: Finite Difference Methods

Our diffusion process is modeled by the equation $u_t = \gamma u_{xx}$. In this section, we shall consider the initial value problem

$$\begin{cases} u_t = \gamma u_{xx} \\ u(x,t)\big|_{t=0} = u_0(x) \end{cases} . \tag{2}$$

In particular, we shall put off discussion of boundary conditions.

To solve this equation, we shall choose a spatial grid size h and a time step k. We then set $x_m = mh$ and $t_n = nk$ and we use the approximation $u_m^n \approx u(x_m, t_n)$.

A *finite-difference* method for solving the equation (2) is a set of equations for the approximations u_m^n. There are a number of different finite difference methods that we can use; below we present a number of methods.

Forward Time, Central Space

Taylor's Theorem applied to the function $t \mapsto u(x,t)$ says that

$$u(x,t+k) = u(x,t) + u_t(x,t)k + O(k^2),$$

so applying this result at the grid point (x_m, t_n), we find that

$$u_t(x_m,t_n) = \frac{u_m^{n+1} - u_m^n}{k} + O(k).$$

Similarly

$$u(x \pm h,t) = u(x,t) \pm u_x(x,t)h + u_{xx}(x,t)\tfrac{h^2}{2} \pm u_{xxx}(x,t)\tfrac{h^3}{6} + O(h^4).$$

Thus, adding these two results at the grid point (x_m, t_n), we find that

$$u_{xx}(x_m,t_n) = \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2} + O(h^2).$$

Thus

$$[u_t - \gamma u_{xx}] - \left[\frac{u_m^{n+1} - u_m^n}{k} - \gamma \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}\right] = O(k) + O(h^2).$$

The Forward Time, Central Space approximation to the diffusion equation is then

$$\frac{u_m^{n+1} - u_m^n}{k} = \gamma \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}. \tag{3}$$

The error in the approximation is $O(h^2) + O(k)$. This is an explicit method in the following sense. From the initial data $u(x,t)\big|_{t=0} = u_0(x)$, we can determine all of the approximations u_m^0. Solving (3), we see that

$$u_m^{n+1} = u_m^n + \left(\frac{\gamma k}{h^2}\right)\left(u_{m+1}^n - 2u_m^n + u_{m-1}^n\right).$$

We can use this equation to find all of the approximations u_m^1, the repeat the process to find u_m^2, u_m^3 and so on.

Backward Time, Central Space

The backward time, central space scheme is

$$\frac{u_m^{n+1} - u_m^n}{k} = \gamma \frac{u_{m+1}^{n+1} - 2u_m^{n+1} + u_{m-1}^{n+1}}{h^2}. \tag{4}$$

The derivation is similar to the derivation for the Forward Time Central Space method and has error $O(h^2) + O(k)$. However unlike that method this is an implicit scheme. Indeed, once u_m^0 is known, we see that (4) is a set of equations for u_m^1 which we have to solve through some method.

Crank-Nicolson (1947)

The Crank-Nicolson scheme is

$$\frac{u_m^{n+1} - u_m^n}{k} = \gamma \left\{ \frac{1}{2} \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2} + \frac{1}{2} \frac{u_{m+1}^{n+1} - 2u_m^{n+1} + u_{m-1}^{n+1}}{h^2} \right\}.$$

This too is an implicit scheme, but with error $O(h^2) + O(k^2)$. Note the higher accuracy!

Leapfrog Scheme

The Leapfrog scheme is

$$\frac{u_m^{n+1} - u_m^{n-1}}{2k} = \gamma \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}.$$

This is a two-step scheme, meaning that the calculation of u_m^{n+1} requires knowledge of the solution at *two* prior times u_m^n and u_m^{n-1}. This causes some difficulty in implementation; indeed to calculate u_m^2, we need to know u_m^1 and u_m^0. Since our scheme does not give us a way to calculate u_m^1, we must use some other method to do so. The error in the Leapfrog scheme is $O(h^2) + O(k^2)$.

DuFort-Frankel

The DuFort Frankel scheme is

$$\frac{u_m^{n+1} - u_m^{n-1}}{2k} = \gamma \frac{u_{m+1}^n - \left(u_m^{n-1} + u_m^{n+1}\right) + u_{m-1}^n}{h^2}.$$

This is similar to the Leapfrog scheme; it is a two-step scheme, with error $O(h^2) + O(k^2)$.

Section 4: Convergence, Consistency, and Stability

We would like to know which of the methods we have presented are good, and which are bad. The concepts used to make that determination actually apply to many more partial differential equations than just the diffusion equation presented here. Define

$$Pu = u_t - \gamma u_{xx};$$

then the diffusion equation simply requires $Pu = 0$.

Convergence

Consider a partial differential equation $Pu = 0$ that is first order in time. A finite difference scheme $P_{k,h} u_m^n = 0$ is *convergent* if given a solution $u(x,t)$ of the partial differential equation with $u(x,0) = u_0(x)$, then given any initial data u_m^0 with $u_m^0 \to u_0(x)$ as $x_m \to x$ we have $u_m^n \to u(x,t)$ as $(x_m, t_n) \to (x,t)$ while $h,k \to 0$.

This is the key property we want our finite difference method to have, because it says that our approximation are close to the actual solution. However, in general, it is very difficult to use the definition to prove that a scheme is convergent.

Consistency

A finite difference scheme $P_{k,h}u_m^n = 0$ is consistent with the partial differential equation $Pu = 0$ if

$$P_{k,h}\phi - P\phi \to 0$$

as $h,k \to 0$ for all smooth functions ϕ. All of the schemes we have discussed are consistent.

Example: For the forward time central space scheme

$$P\phi = \phi_t - \gamma\phi_{xx},$$

while

$$P_{k,h}\phi = \frac{\phi_m^{n+1} - \phi_m^n}{k} - \gamma\frac{\phi_{m+1}^n - 2\phi_m^n + \phi_{m-1}^n}{h^2}.$$

Thus

$$P_{k,h}\phi - P\phi = \left(\phi_t - \gamma\phi_{xx} + O(k) + O(h^2)\right) - \left(\phi_t - \gamma\phi_{xx}\right) = O(k) + O(h^2) \to 0.$$

Consistent schemes do not need to be convergent! Though we want all of our schemes to be consistent, this is not sufficient.

Example: Consider the initial value problem

$$\begin{cases} u_t + u_x = 0 \\ u|_{t=0} = f(x) \end{cases}.$$

This has solution

$$u(x,t) = f(x-t)$$

which can be verified by substitution. The solution at time t is just the initial data translated to the right by a distance t.

The scheme

$$\frac{u_m^{n+1} - u_m^n}{k} + \frac{u_{m+1}^n - u_m^n}{h} = 0$$

is a consistent scheme with error $O(h) + O(k)$. Note however, that this is equivalent to

$$u_m^{n+1} = u_m^n - \tfrac{k}{h}\left(u_{m+1}^n - u_m^n\right)$$

so that the solution at time t_{n+1} depends only on the value of the solution at time t_n at the same point and points to the right; thus the scheme moves information to the left.

If

$$f(x) = \begin{cases} 0 & x > 0 \\ 1 & x < 0 \end{cases},$$

then

$$u(x,t) = \begin{cases} 0 & x > t \\ 1 & x < t \end{cases}.$$

However

$$u_m^n = 0$$

for $x_m > 0$, for all time t_n.

Stability

A finite difference scheme $P_{k,h}u_m^n = 0$ for a (first-order in time) partial differential equation is *stable* if there are numbers J, h_0, and k_0 so that for every time T there is a constant C_T with

$$h\sum_m \left|u_m^n\right|^2 \le C_T h \sum_{j=0}^{J} \sum_m \left|u_m^j\right|^2$$

where $0 \le nk \le T$, $0 < h \le h_0$, and $0 < k \le k_0$.

Roughly, this says that the size of the solution at time t_n is bounded by an multiple of the size of the solution at t_0, t_1, ..., t_J. This is related to the concept for solutions of partial differential equations called well-posedness.

The initial value problem for the (first order in time) partial differential equation $Pu = 0$ is *well-posed* if for all T there is a constant C_T so that every solution satisfies

$$\int_{-\infty}^{\infty} \left|u(x,t)\right|^2 dx \le C_T \int_{-\infty}^{\infty} \left|u(x,0)\right|^2 dx$$

for $0 \le t \le T$.

This says that the size of the solution at time t is no larger than a multiple of the size of the solution at time $t = 0$.

The diffusion equation is well posed.

Lax-Richtmyer Equivalence Theorem

The key method used to determine if a finite difference method is convergent is the Lax-Richtmyer Equivalence Theorem.

A consistent finite difference scheme for a (first order in time) partial differential equation for which the initial value problem is well posed is convergent if and only if it is stable.

We have already seen that checking consistency of a finite difference method is not too difficult; if we can find a simple way to check stability, then we will be able to determine is a finite difference method is convergent.

Section 5: Von Neumann Analysis

The idea of Von Neumann analysis is that, to determine the stability of a finite difference method, it is sufficient to analyze trigonometric functions.

In particular, to determine if a finite difference scheme is stable, we simply need to analyze how the scheme acts on trigonometric functions. We look for solutions of the form

$$u_m^n = g^n e^{im\theta}.$$

[Recall $e^{i\theta} = \cos\theta + i\sin\theta$.] The function $g(\theta)$ is a complex valued function called the amplification factor for the method. We can determine g by substituting this form into the finite difference method. In a one step scheme there is one solution, while in a two-step scheme there are two solutions.

- In a one step scheme, the scheme is stable if and only if $|g| \le 1$.
- In a two-step scheme, the scheme is stable if each of the roots $g_\pm$ satisfies $|g_\pm| \le 1$, and at least one of the roots satisfies $|g| < 1$.

Example: The forward time, central space scheme is

$$\frac{u_m^{n+1} - u_m^n}{k} = \gamma \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}.$$

This can be written as

$$u_m^{n+1} = u_m^n + \lambda\left(u_{m+1}^n - 2u_m^n + u_{m-1}^n\right)$$

where $\lambda = \frac{\gamma k}{h^2}$. Substituting, we find that

$$g^{n+1}e^{im\theta} = g^n e^{im\theta} + \lambda\left(g^n e^{i(m+1)\theta} - 2g^n e^{im\theta} + g^n e^{i(m-1)\theta}\right)$$

so

$$g = 1 + \lambda\left(e^{i\theta} - 2 + e^{-i\theta}\right).$$

Now

$$e^{i\theta} - 2 + e^{-i\theta} = 2\cos\theta - 2 = -4\left(\frac{1-\cos\theta}{2}\right) = -4\sin^2\frac{\theta}{2}$$

so that

$$g = 1 - 4\lambda\sin^2\frac{\theta}{2}.$$

To have stability, we need $|\, g\, | \le 1$, or equivalently

$$-1 \le 1 - 4\lambda\sin^2\tfrac{\theta}{2} \le 1,$$

which means

$$-2 \le -4\lambda \sin^2 \tfrac{\theta}{2} \le 0.$$

Thus we have stability if and only if

$$\lambda \sin^2 \tfrac{\theta}{2} \le \tfrac{1}{2}$$

for all θ; this is equivalent to the restriction that

$$\lambda = \frac{\gamma k}{h^2} \le \tfrac{1}{2}.$$

This restriction is problematic in practice. Suppose that $\gamma = 1$, and $L = 1$. If we were to use 100 grid points in space, we would then need to set $h = \frac{1}{100} = 0.01$. To use Forward Time Central Space, we must have $\lambda \le \frac{1}{2}$, which requires $\frac{\gamma k}{h^2} \le \frac{1}{2}$. This simplifies to the requirement that $k \le \frac{1}{2}h^2 = 0.00005$. This has the practical effect of (dramatically) increasing the time needed to perform a computation. If we wanted to approximate the value of the solution when $t = 1$, we will need to perform 20,000 time steps in forward time central space.

In the exercises, you will show that the Backward Time Central Space method and the Crank-Nicolson method are stable for all values of λ, while the Leapfrog method is unstable for all values of λ. Because the Backward Time Central Space method and the Crank-Nicolson have no stability restrictions on the step size, they are more suited for practical use. The downside to these methods is that they are implicit, which makes them more difficult to code.

Section 6: Boundary Conditions

As we saw in Section 2, to solve the diffusion equation, we need initial data and boundary data. In this section we shall discuss the different types of boundary data that can be given, and how they affect our numerical methods for computing the solution.

We shall impose one boundary condition at each end- at $x = 0$ and $x = L$, however the conditions do not need to be of the same type. To simplify the discussion below, we shall assume that we are imposing the condition at $x = 0$.

Dirichlet Conditions

A Dirichlet condition at $x = 0$ takes the form

$$u(0,t) = A \text{ for all } t.$$

If we are using the diffusion equation to model heat flow, then this condition means that the boundary is kept at a fixed temperature. In general, the value A can be allowed to depend on time.

This condition is simply implemented by setting $u_0^n = A$ for all t at each time.

Neumann Conditions.

A Neumann condition at $x = 0$ takes the form

$$u_x(0,t) = A \text{ for all } t.$$

If we are using the diffusion equation to model heat flow, and if $u_x(0,t) = 0$ for all t, the boundary is insulated, meaning that heat does not flow across the boundary. This is also called the *adiabatic* case. In general, if $u_x(0,t) = A$ for all t, the heat flux across the boundary is prescribed.

To implement this condition, we could use the approximation

$$\frac{\partial u}{\partial x}(0,t_n) = \frac{u_1^n - u_0^n}{h} + O(h)$$

to determine u_0^n. However, this method is only $O(h)$ accurate, and will degrade schemes like forward-time central-space, and Crank-Nicolson; thus it is not used.

Instead, there are two different methods that are encountered in practice. First, we can use the approximation

$$\frac{\partial u}{\partial x}(0,t_n) = \frac{-3u_0^n + 4u_1^n - u_2^n}{h} + O(h^2)$$

to determine u_0^n. This method is $O(h^2)$ accurate.

In the second method, we assume that there is a hypothetical grid point at x_{-1}, and use the second order approximation

$$\frac{\partial u}{\partial x}(0,t_n) = \frac{u_1^n - u_{-1}^n}{2h} + O(h^2).$$

By applying the finite difference method used to solve the equation the at the grid point x_0, we obtain another equation for u_{-1}^n, allowing us to eliminate u_{-1}^n. For example, if $u_x(0,t) = 0$, and we are using the forward-time, central-space scheme, then

$$u_0^{n+1} = u_0^n + \left(\frac{\gamma k}{h^2}\right)\left(u_1^n - 2u_0^n + u_{-1}^n\right)$$

so setting $\lambda = \gamma k / h^2$ and substituting

$$\frac{u_1^n - u_{-1}^n}{2h} = \frac{\partial u}{\partial x}(0,t_n) = 0$$

we see that

$$\begin{aligned} u_0^{n+1} &= u_0^n + \lambda\left(u_1^n - 2u_0^n + u_1^n\right) \\ &= (1-2\lambda)u_0^n + 2\lambda u_1^n. \end{aligned}$$

Thus, knowing the values of u_m^n, we can determine the value of the boundary datum u_0^{n+1}.

Boundary Conditions of the third kind

At $x = 0$, these conditions have the form

$$u_x(0,t) - cu(0,t) = 0$$

or

$$u_x(0,t) = c[u(0,t) - U].$$

These conditions arise because it is impossible to obtain pure Dirichlet or pure Neumann conditions in practice.

Use any of the techniques described for Neumann conditions to approximate u_x.

Section 7: The Thomas Algorithm

The Backward-Time Central-Space scheme and the Crank-Nicolson scheme are implicit schemes, because they do not give us an explicit formula for the values of u^{n+1} in terms of the different values of u^n, but rather give us an equation that the u^{n+1} must solve. In this section, we shall examine equations of this type.

Suppose that we have $M+1$ unknowns $u_0, u_1, \dots, u_M$, and suppose that we know that they satisfy the $M-1$ equations

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i \qquad 1 \le i \le M-1 \qquad (5)$$

together with two other equations.

To solve this system, we shall use the Thomas algorithm. We look for a relationship among the variables u_i of the form

$$u_i = p_i u_{i+1} + q_i \qquad 0 \le i \le M-1 \qquad (6)$$

where the values p_i and q_i are to be determined. Substitute this into (5) to see that

$$a_i(p_{i-1}u_i + q_{i-1}) + b_i u_i + c_i u_{i+1} = d_i.$$

Thus

$$(a_i p_{i-1} + b_i)u_i + c_i u_{i+1} + a_i q_{i-1} = d_i$$

or

$$u_i = \frac{-c_i}{a_i p_{i-1} + b_i} u_{i+1} + \frac{d_i - a_i q_{i-1}}{a_i p_{i-1} + b_i}.$$

Comparing this with (6), we see that

$$\begin{cases} p_i = \dfrac{-c_i}{a_i p_{i-1} + b_i} \\ q_i = \dfrac{d_i - a_i q_{i-1}}{a_i p_{i-1} + b_i} \end{cases} \qquad 1 \le i \le M-1. \qquad (7)$$

Thus, if we know p_0 and q_0, we can use (7) to determine p_i and q_i for $1 \le i \le M-1$.

Dirichlet Data

How we proceed, now depends on which other two equations we add to (5). For our first case, let us suppose that

$$\begin{cases} u_0 = \beta_0 \\ u_M = \beta_M \end{cases}.$$

Because $u_0 = \beta_0$, then (6) for $i = 0$ which is $u_0 = p_0 u_1 + q_0$ has the solution $p_0 = 0$, $q_0 = \beta_0$. We then use (7) to determine p_i and q_i for $1 \le i \le M-1$. Then to determine the solution, we use (6), starting with the fact that we know $u_M = \beta_M$. In particular, because p_{M-1} and q_{M-1} are now known, we can calculate $u_{M-1} = p_{M-1}\beta_M + q_M$ and then continue inductively.

Neumann Data

As a second case, suppose that

$$\begin{cases} \qquad\qquad b_0 u_0 + c_0 u_1 = d_0 \\ a_M u_{M-1} + b_M u_M \qquad = d_M \end{cases}.$$

The first of these can be written as

$$u_0 = \frac{-c_0}{b_0} u_1 + \frac{d_0}{b_0}$$

so we set $p_0 = \dfrac{-c_0}{b_0}$ and $q_0 = \dfrac{d_0}{b_0}$. To determine u_M, we substitute (6) into the second of our equations to see that

$$a_M \left(p_{M-1} u_M + q_{M-1} \right) + b_M u_M = d_M$$

so that

$$u_M = \frac{d_M - a_M q_{M-1}}{a_M p_{M-1} + b_M}.$$

We handle the mixed cases in the same fashion.

The Thomas algorithm is not a good choice for every tridiagonal system. In particular, if $|p_i| > 1$, then this algorithm will magnify errors every time it is run. One condition which ensures that $|p_i| \le 1$ is diagonal dominance. This is the requirement that $|a_i| + |c_i| \le |b_i|$. This condition should be checked before this algorithm is used.

Assignments

1. Verify that, if let $u_i(t)$ be the concentration of smog at the point x_i at time t and if the rate of change of the smog concentration at a point is proportional to the difference between the concentrations at adjacent points, then $u_i'(t) = c[u_{i+1} - 2u_i + u_{i-1}]$.

2. Let $f(x)$ be a continuous function defined on the interval $[0, L]$. We shall prove that if $\int_a^b f(x)\,dx = 0$ for every $(a,b) \subset [0,L]$, then $f(x) = 0$.

a. Show that if $z \in (0, L)$, and $f(z) > 0$, then there exists a number $\varepsilon > 0$ so that for all $x \in (z - \varepsilon, z + \varepsilon)$ we have $f(x) > \frac{1}{2} f(z) > 0$. [Hint: What is the definition of a continuous function?]

b. Suppose that $f(z) > 0$. Show that there is an interval $(a,b) \subset [0,L]$ so that $\int_a^b f(x)\,dx > 0$.

c. Suppose that $f(z) < 0$. Show that there is an interval $(a,b) \subset [0,L]$ so that $\int_a^b f(x)\,dx < 0$.

d. Suppose that $f(z) \ne 0$. Show that there is an interval $(a,b) \subset [0,L]$ so that $\int_a^b f(x)\,dx \ne 0$. Conclude.

3. Show that the heat equation is well posed. [Hint: Multiply the equation by $u(x,t)$, and integrate in x and t. Use integration by parts. You can assume that $u(x,t) \to 0$ as $x \to \infty$, for every t.]

4. Show that the Leapfrog scheme is consistent, with accuracy $O(h^2) + O(k^2)$.

5. Show that the Crank-Nicolson scheme is consistent, with accuracy $O(h^2) + O(k^2)$. [Hint: Use the fact that $u_{tt} = \gamma u_{xxt}$ and $u_{xx}(x_m, t_{n+1}) = u_{xx}(x_m, t_n) + k u_{xxt} + O(k^2)$.]

6. Show that the Backward Time Central Space method is stable for all $\lambda = \dfrac{\gamma k}{h^2}$.

7. Show that the Crank-Nicolson scheme is stable for all $\lambda = \dfrac{\gamma k}{h^2}$.

8. Show that the Leapfrog scheme is unstable for all $\lambda = \dfrac{\gamma k}{h^2}$.

9. Prove the approximation $\frac{\partial u}{\partial x}(0,t_n) = \frac{-3u_0^n + 4u_1^n - u_2^n}{h} + O(h^2)$.

10. Suppose that $u_x(0,t) = A$. Use the approximation $\frac{\partial u}{\partial x}(0,t_n) = \frac{u_1^n - u_{-1}^n}{2h} + O(h^2)$ and the forward-time central-space scheme to obtain an equation for u_0^{n+1} in terms of u_0^n, u_1^n, and the parameters A, h, and λ.

Project

Write a C++ program that simulates the diffusion of heat in one dimension.

As input, the program should take

- The type of boundary data (Dirichlet or Neumann) at each endpoint and its value.
- The diffusion coefficient.
- The user should be able to use the mouse to determine the initial data.
- The step size for the simulation.
- The total time for the simulation.

As output, the program should return

- A graphical representation of temperature of the object.

The user should be able to select which method is used to compute the solution- either

- Forward-Time Central-Space, or
- Backward-Time Central-Space.

The program should be written using good object oriented programming techniques.

You are then to answer the following questions:

1. Consider a 60 cm long steel rod (with $\gamma = 0.15$ cm^2/s). Suppose that the ends are kept at 0^o, and that initially the first 30 cm of the rod are at 100^o, while the last 30 cm of the rod are at 0^o. Use your simulation to determine the temperature of the rod after 10 minutes have elapsed. What is the value of the temperature at the center of the rod?
2. Consider the same steel rod. Suppose that heat is supplied through the left end of the rod at the rate 10^o/m, while the right end is kept at 0^o. Suppose that initially the entire rod is at 0^o. Use your simulation to determine the temperature of the rod after 10 minutes have elapsed. What is the value of the temperature at the center of the rod?

Write a good report of your findings, including

- What is the mathematical model of the problem?
- What is the numerical method used to solve the problem?
- What is the structure of your program?

When answering these questions, it is ***essential*** that you address the question of how the choice of step size affects the result. You ***must*** run the simulation for different step sizes to draw correct conclusions.

Timers

Section 1: Introduction

There are times when a programmer wants a piece of code to be executed at particular times. One example would be a program that displays a clock; we would want to update the clock's display once each second. A second example would be a program with animated graphics. Rather than have the graphics redrawn as often as the code permits, we would instead like to have the graphics drawn at some predetermined rate, say 20 frames per second.

To illustrate these ideas, we shall write a short program that rotates a ball across the screen at a smooth rate of 20 frames per second. At the same time, the program will display the current time.

Section 2: The Skeleton

We begin with a dialog based program without About box called Timers. We shall remove the Cancel box, and modify the OK box to read Exit Program. Next, we add a button called Draw with an ID of `IDC_DRAW`. We remove the default static text, and add two static text boxes. The first says "The current time is " while the second simply says "Current Time". The ID for the first of these can remain in its default setting of `IDC_STATIC`. However, because we want to modify the text in the second box to give the current time, we must give it a

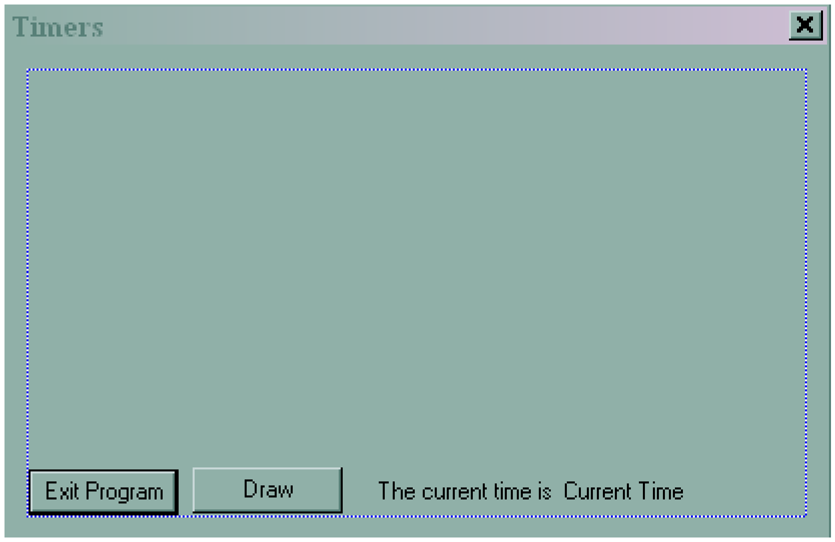


Figure 1: The Timers dialog box

unique ID. In our case we use `IDC_CURRENT_TIME`. At this point, our dialog box looks like Figure 1.

Our program will draw a ball on the screen, and move it in a circle when the Start button is pressed. Thus we need to add a variable to keep track of the position of the ball, we must initialize it, and we must modify the `OnPaint()` method to draw it. We begin by adding the private variable `m_ptBall` of type `CPoint`. Because we want the ball to move in a circle, we shall also add a private variable `m_ptCenter` of type `CPoint` which will store the center point of our dialog box, and will be used as the center of the circle that the moving ball will traverse. We initialize these variables in the `OnInitDialog()` method with the following code

```
BOOL CTimersDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
         automatically
      //  when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);             // Set big icon
      SetIcon(m_hIcon, FALSE);            // Set small icon

      // TODO: Add extra initialization here

      CRect CurrentRect;
      GetClientRect(CurrentRect);
      m_ptCenter = CurrentRect.CenterPoint();
      m_ptBall = m_ptCenter + CPoint(50,0);

      return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

Next we need to add code to the OnPaint() method. We can do this with the following code

```
void CTimersDlg::OnPaint()
{
      if (IsIconic())
      {
            CPaintDC dc(this); // device context for painting

            SendMessage(WM_ICONERASEBKGND, (WPARAM)
                  dc.GetSafeHdc(), 0);

            // Center icon in client rectangle
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
            GetClientRect(&rect);
            int x = (rect.Width() - cxIcon + 1) / 2;
            int y = (rect.Height() - cyIcon + 1) / 2;

            // Draw the icon
            dc.DrawIcon(x, y, m_hIcon);
      }
      else
      {

            CPaintDC dc(this);

            CBrush* OldBrush;
            CBrush PurpleBrush;
            PurpleBrush.CreateSolidBrush(RGB(128,0,255));
            OldBrush = dc.SelectObject(&PurpleBrush);

            CRect Ball(m_ptBall,m_ptBall);
            Ball.InflateRect(10,10);
            dc.Ellipse(Ball);

            dc.SelectObject(OldBrush);


            CDialog::OnPaint();
      }
}
```

Only the portion of the code in the box is new; the rest was inserted by the MFC AppWizard when the program skeleton was first created.

To have our ball move on the screen, we need to add some code to the Start button. We call that function OnButtonDraw(), and start with the following code

```
void CTimersDlg::OnButtonDraw()
{
      // TODO: Add your control notification handler code here

      double width = (double)(m_ptBall.x - m_ptCenter.x);

      double x;
      double y;
      const double pi = 4.0*atan(1.0);

      double count = 500;     //Draw ball 500 times per circuit
      for(int i=0; i<= 4.0*count; i++)    //Make 4 circuits
      {
            x = cos(2.0*pi*(double)(i)/count);
            y = sin(2.0*pi*(double)(i)/count);

            m_ptBall = m_ptCenter + CPoint( (int)(width * x),
                  (int)(width*y) );

            Invalidate();
            OnPaint();
      }

}
```

Note that this code requires that you include <math.h>.

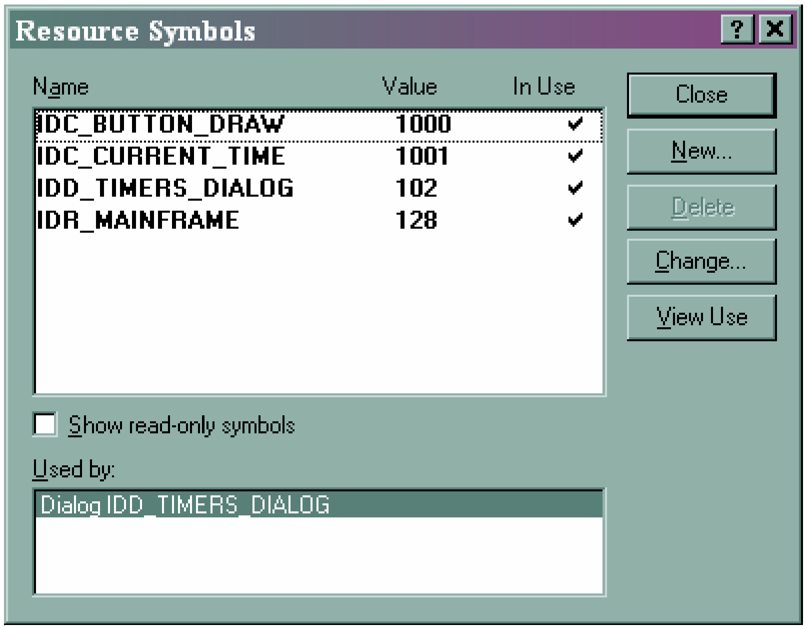
At this point, we can compile and run our program. When the start button is pressed, the ball will make four complete circles about the center of our dialog box, but the motion is choppy and flickers a great deal. One reason for this is that, our code attempts to redraw the screen every time through the loop. Since the loop is not terribly complex, this occurs very rapidly. Hence the code spends most of its time drawing and erasing balls on the screen. On my machine, the loop takes about 2 seconds to execute, so we attempt to redraw the screen 500 times in two seconds- yet my monitor is set at 85 Hz, so the monitor will only refresh 170 times in two seconds. Thus, it is not surprising that the image flickers. To control this problem, we shall rewrite the code so that we only attempt to draw the screen 20 times per second. We begin by removing the lines

```
Invalidate();
OnPaint();
```

from our `OnDrawButton()` code.

Section 3: Timers

A timer is a resource like a dialog box, however the process of adding a timer resource to a program is somewhat different. From the main menu, choose View, then Resource Symbols. Equivalently, you can go to the Resource Tab, then right-click on Timers Resources, and select Resource Symbols. In either case you are presented with a dialog box like that in Figure 2. We then add a new resource



Name	Value	In Use
IDC_BUTTON_DRAW	1000	✓
IDC_CURRENT_TIME	1001	✓
IDD_TIMERS_DIALOG	102	✓
IDR_MAINFRAME	128	✓

Dialog IDD_TIMERS_DIALOG

Figure 2: The Resource Symbols menu

symbol called ID_TIMER. You can use any unused positive integer for the value, we selected 101.

To make use of this timer, we must initialize it. We can do so in the `OnInitDialog()` method of our `CTimersDlg` class by adding the boxed code below.

```
BOOL CTimersDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
automatically
      //  when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);            // Set big icon
      SetIcon(m_hIcon, FALSE);           // Set small icon

      // TODO: Add extra initialization here

      CRect CurrentRect;
      GetClientRect(CurrentRect);
      m_ptCenter = CurrentRect.CenterPoint();
      m_ptBall = m_ptCenter + CPoint(50,0);

      SetTimer(ID_TIMER,50,NULL);

return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

The function `SetTimer` is a member of `CWnd`, which is inherited by `CDialog` and hence by `CTimersDlg`. It takes three parameters. The first is the ID of a timer; in our case this is `ID_TIMER`. The second parameter is the time, in milliseconds, between timer messages. We set our to 50 milliseconds so that it fires 20 times each second. Last is a pointer to a callback function; if this is left `NULL` as in our case, the messages are handled by the usual message queue.

Timers are a limited resource, and only a small number are available for use. The `SetTimer` function returns an integer value; if it is zero the attempt to set the timer was unsuccessful.

We have set our timer to use the usual message queue; this means that every 50 milliseconds a `WM_TIMER` message will be sent. We can then construct a handler for this message and write code that will be executed at predetermined times. There is one caveat to this process however; the `WM_TIMER` message will not be sent if our program’s message queue has unprocessed messages. We will see the effect of this later.

We can now add a message handler for the `WM_TIMER` message. It is called `OnTimer` and its default code is

```
void CTimersDlg::OnTimer(UINT nIDEvent)
{
      // TODO: Add your message handler code here and/or call
default

      CDialog::OnTimer(nIDEvent);
}
```

The one parameter it receives, `nIDEvent` is identifier of the timer. In our case since we are only using one timer, this parameter will not be needed.

Section 4: The Clock

To see how we can use the `WM_TIMER` messages, we will modify our program to display the current time. To do so, we first use the Class Wizard to associate a variable to the static text with ID of `IDC_CURRENT_TIME` where our time will be displayed. This will be a variable of type `CString` called `m_strTime`.

The class `CString` is a class in MFC for handling strings. It supports most of the common string operations. If `m_strString` is a `CString` variable, you can assign it a value simply by the code

```
m_strString = "This is the value of my string"
```

You can concatenate strings using the + operator. To convert a number to a string, we use the `format` function.

The `format` function takes two or more arguments. The last are the values that are to be converted into a string, while the first, enclosed in quotes, is the formatting of the string using the standard format specifications for the ANSI C functions `printf` and `wprintf`. There are different codes that describe the type of variable:

- `%d` or `%i` indicate a signed decimal integer,
- `%u` indicates an unsigned decimal integer,
- `%f` is for doubles in the form 123.456, and
- `%e` is for a double in the form 1.23456e+02.

In each case, the width of the result can be specified by an integer before the letter code. Thus `%3d` returns a signed integer of width 3. If the leading number is a zero, then the width will be filled by leading zeros. These codes can be separated by any characters; the character % is indicated by `%%`.

Examples:

Code	Output
`Format("%d:%d",10,1)`	10:1
`Format("%d %02d",10,1)`	10 01
`Format("%d text! %02d",10,1)`	10 text! 01
`Format("%d %02d:%d",12,3,90)`	12 03:90
`Format("%d %02d:%.2f",12,3,90.1)`	12 03:90.10
`Format("%d%% %02d - %.3e",12,3,90.1)`	12% 03 – 9.010e+001

To display the current time, we need to first obtain the current time. We can do this by using the `GetCurrentTime` function of the `CTime` class. Because this is a static member function, we use the following code to initialize it

```
        CTime curTime = CTime::GetCurrentTime();
```

This puts the current time into the `CTime` variable `curTime`. The `CTime` methods `GetHour()`, `GetMinute()` and `GetSecond()` return integers that contain the hour, minute, and second recorded in that `CTime` object.

We can then display the result on the screen by using the following code.

```
void CTimersDlg::OnTimer(UINT nIDEvent)
{
      // TODO: Add your message handler code here and/or call
default

      CTime curTime = CTime::GetCurrentTime();

      m_strTime.Format("%2d:%02d:%02d",curTime.GetHour(),
            curTime.GetMinute(),curTime.GetSecond());
      UpdateData(FALSE);

      CDialog::OnTimer(nIDEvent);
}
```

Note the formatting that was used in the string, as well as the usual `UpdateData(FALSE)` command to update the values on the screen.

At this point, if we compile and run the program, it will keep the current time. Next, we modify the code that revolves the ball to eliminate the flicker we have already seen.

Section 5: Using Timers for Animation

We would like to use our timer to redraw the screen each time it is called. Because only a portion of the screen needs to be redrawn each time, we shall only redraw the portions of the screen where the ball is to appear and disappear. To do this, we add a new private variable of type `CPoint` called `m_ptLastDrawnCenter`. This will keep track of the `CPoint` of the center of the ball the last time it has been drawn on the screen. We initialize it to be the same as `m_ptCenter` in the `OnInitDialog()` routine, which now reads as follows.

```
BOOL CTimersDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Set the icon for this dialog.  The framework does this
      automatically
      //  when the application's main window is not a dialog
      SetIcon(m_hIcon, TRUE);              // Set big icon
      SetIcon(m_hIcon, FALSE);             // Set small icon

      // TODO: Add extra initialization here

      CRect CurrentRect;
      GetClientRect(CurrentRect);
      m_ptCenter = CurrentRect.CenterPoint();
      m_ptBall = m_ptCenter + CPoint(50,0);
      m_ptLastDrawnBall = m_ptBall;
      SetTimer(ID_TIMER,50,NULL);


      return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

We then modify the `OnTimer()` function to first check to see if the ball has moved. If so, it determines the area of the screen previously occupied by the ball and the area of the screen that the ball will occupy. It then uses `InvalidateRect()` to ensure that these areas are redrawn, then calls `OnPaint()`. The resulting `OnTimer()` function then reads as follows.

```
void CTimersDlg::OnTimer(UINT nIDEvent)
{
      // TODO: Add your message handler code here and/or call
default

      if(m_ptBall != m_ptLastDrawnBall)
      {
            CRect OldBall(m_ptLastDrawnBall,
                    m_ptLastDrawnBall);
            CRect CurrentBall(m_ptBall,m_ptBall);

            OldBall.InflateRect(10,10);
            CurrentBall.InflateRect(10,10);

            InvalidateRect(OldBall);
            InvalidateRect(CurrentBall);

            m_ptLastDrawnBall = m_ptBall;
            OnPaint();
      }


      CTime curTime = CTime::GetCurrentTime();

      m_strTime.Format("%2d:%02d:%02d",curTime.GetHour(),curTime.
GetMinute(),curTime.GetSecond());
      UpdateData(FALSE);

      CDialog::OnTimer(nIDEvent);
}
```

The new material is boxed.

One might expect that these are all of the changes that need to be made. Compiling and executing the code, you will find however, that the ball will not move when the Draw button is pressed. The see what has occurred, let us modify the number of iterations in our main loop; increasing it to 500,000 from 500. This gives us the following code for `OnButtonDraw()`

```
void CTimersDlg::OnButtonDraw()
{
      // TODO: Add your control notification handler code here

      double width = (double)(m_ptBall.x - m_ptCenter.x);

      double x;
      double y;
      const double pi = 4.0*atan(1.0);

      double count = 500000;
      for(int i=0; i<= 4.0*count; i++)
      {
            x = cos(2.0*pi*(double)(i)/count);
            y = sin(2.0*pi*(double)(i)/count);

            m_ptBall = m_ptCenter + CPoint( (int)(width * x),
            (int)(width*y) );

//          Invalidate();           We now use timers to control
//          OnPaint();              the drawing process
      }


}
```

When this code is run, you will see that the clock will pause momentarily when the Draw button is pressed, and increasing the value of `count` increases the length of the pause.

The only explanation for this behavior is that the `OnTimer()` method is not being called. How can this occur? Recall that the `WM_TIMER` message will not be sent if there is another message waiting in the applications message queue. What happens is that the program waits to finish executing the `OnButtonDraw()` code before pulling another message off the queue. This leaves an uncalled `WM_TIMER` message on the queue, and no new `WM_TIMER` messages will be sent until `OnButtonDraw()` completes.

How can we avoid this situation? The solution is to add code to the `OnButtonDraw()` function to determine if there are new messages in the queue. If so, these should be handled before the loop continues. We can do this with the following code.

```
void CTimersDlg::OnButtonDraw()
{
      // TODO: Add your control notification handler code here

      double width = (double)(m_ptBall.x - m_ptCenter.x);

      double x;
      double y;
      const double pi = 4.0*atan(1.0);

      MSG msg;

      double count = 500000;
      for(int i=0; i<= 4.0*count; i++)
      {
            x = cos(2.0*pi*(double)(i)/count);
            y = sin(2.0*pi*(double)(i)/count);

            m_ptBall = m_ptCenter + CPoint( (int)(width * x),
            (int)(width*y) );

            while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }

//          Invalidate();             We now use timers to control
//          OnPaint();                the drawing process
      }

}
```

The variable type `MSG` represents a system message. The precise nature of the message structure does not concern us here; details can be found in the help system.

The PeekMessage command has the following general structure

```
BOOL PeekMessage(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT
wMsgFilterMax, UINT wRemoveMsg)      );
```

The variable `lpMSG` is a pointer to an `MSG` variable; if a message is in the queue, it will be stored here. The variable `hWnd` is the window whose message queue is to be examined. If `hWnd` is `NULL` then `PeekMessage` returns all of the messages available to the program. The variables `wMsgFilterMin` and `wMsgFilter` filter the available messages based on their order of arrival; if both of these are zero then no filtering is performed. The last parameter determines the ultimate fate of the message; if it is set to `PM_REMOVE` then the message is removed from the queue after it is read by `PeekMessage`.

The `TranslateMessage` and `DispatchMessage` functions are aptly named, and we shall not delve into their details. Together, they take the message stored in `msg` and deliver it. In particular, they ensure that our `WM_TIMER` messages are delivered, and that the `OnTimer()` function is called.

Section 6: Enabling and Disabling Buttons

There is one difficulty with this approach. The button Draw starts code by sending a message. Suppose that a user presses the Draw button. Then, while the animation is occurring, the user presses the Draw button once again. That message will be received and sent on its way by our code

```
while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
{
      TranslateMessage(&msg);
      DispatchMessage(&msg);
```

This will cause execution to pass to the beginning of the `OnDrawButton` function. This will occur despite the fact that the first pass through the `OnDrawButton` function has not yet terminated. This will cause unpredictable behavior. Similarly, if the End Program button is pressed, the program will attempt to terminate; when this process is completed, the remainder of the `OnDrawButton` function will execute, again causing unpredictable behavior.

To solve this problem, we shall disable the Draw button and Exit Program button while the animation is occurring. To do so, we use the method `GetDlgItem` which is inherited from `CWnd`. Given an ID, it returns a pointer to the `CWnd` object that displays it. The elements of a dialog box are treated as windows and derived from `CWnd`. Thus, we can use the members of `CWnd` to enable and disable the buttons. The resulting code is below; the new material is boxed.

```
void CTimersDlg::OnButtonDraw()
{
      // TODO: Add your control notification handler code here

      double width = (double)(m_ptBall.x - m_ptCenter.x);

      double x;
      double y;
      const double pi = 4.0*atan(1.0);

      MSG msg;

      GetDlgItem(IDC_BUTTON_DRAW)->EnableWindow(FALSE);
      GetDlgItem(IDOK)->EnableWindow(FALSE);

      double count = 5000000;
      for(int i=0; i<= 4.0*count; i++)
      {
            x = cos(2.0*pi*(double)(i)/count);
            y = sin(2.0*pi*(double)(i)/count);

            m_ptBall = m_ptCenter + CPoint( (int)(width * x),
            (int)(width*y) );

            while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }

//          Invalidate();           We now use timers to control
//          OnPaint();              the drawing process
      }

      GetDlgItem(IDC_BUTTON_DRAW)->EnableWindow(TRUE);
      GetDlgItem(IDOK)->EnableWindow(TRUE);

}
```

This technique is very general and very powerful, and can be used to modify the elements of a dialog box during run time.

Assignments

1. Though the code in Section 6 disables the Draw button and the Exit Program button, there are other messages that can still be sent to the program that will cause unpredictable behavior. What are they?

2. What is a multi-threaded program? How do the concepts of a multi-threaded program apply to our program? [Hint: This will require further reading!]

Project: Waves

Section 1: Introduction

How do waves propagate? To understand the answer to this question, we begin by asking what is a wave. There are a number of types of waves, but we shall be interested in two main types:

- Compression Waves,
- Transverse Waves.

An example of a transverse wave is the vibration of a string.

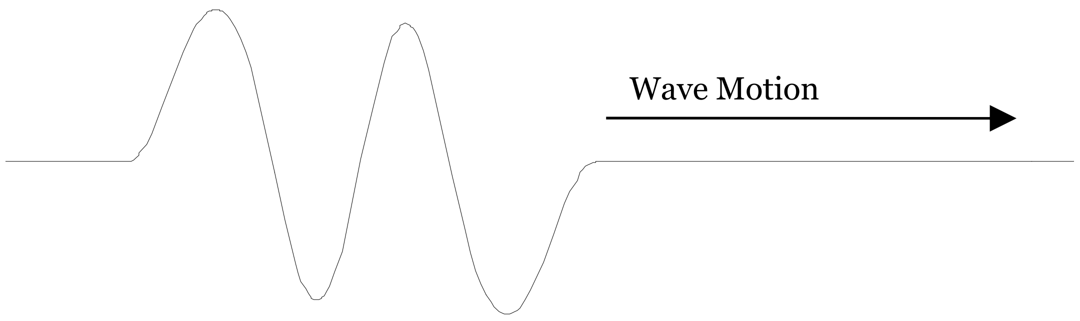


Figure 1: A transverse wave

The wave moves from left to right, but the string itself moves up and down.

A compression wave is also called a longitudinal wave, because the oscillation is in the same direction as the direction of motion.

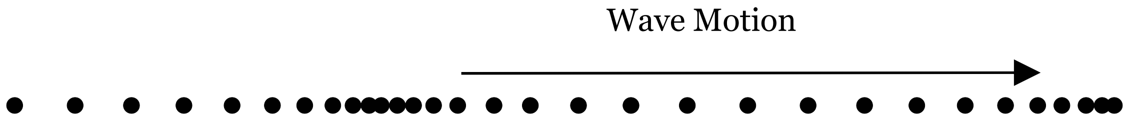


Figure 2: A compression wave

The wave moves from left to right, and the individual components also move from left to right.

We shall construct a model of a compression wave and another model of a transverse wave. We shall then learn how to simulate each.

Section 2: Compression Waves

We begin by constructing a model of a type of compression wave. Consider a long thin rod. If we hold the rod in place, and then strike the left end, the material at the left end of the rod moves towards the right, along the rod. This

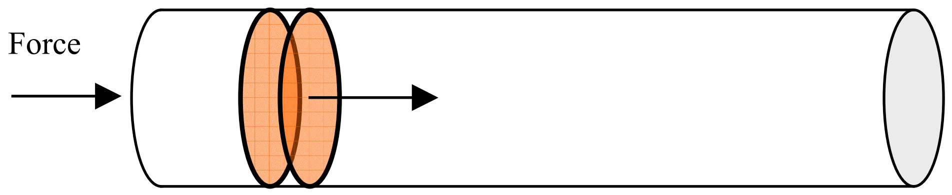


Figure 3: The effect of striking a rod on its end

creates an area of increased density at the left of the rod, and this area travels down the length of the rod. The result is a compression wave.

To create our model, let us imagine that the rod is composed of a large number of identical flat discs. We then need to decide two things-

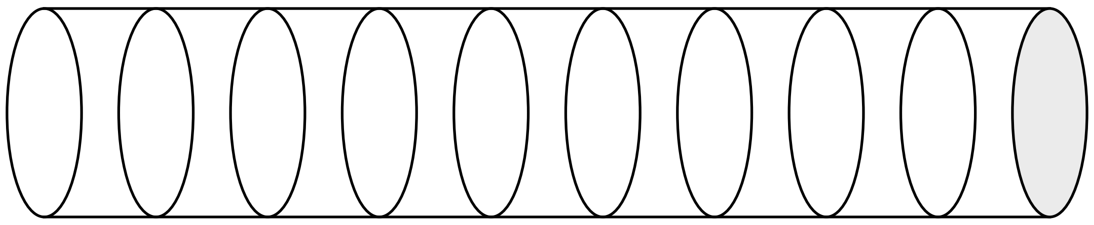


Figure 4: The disc model of the rod

- How does each disc behave?
- What is the relationship between the discs?

Because we want our model to be as simple as possible, let us suppose that the only thing that our individual disc can do is to move from side to side. An area of high pressure will be an area where the discs are clustered close together, while an area of low pressure will be modeled by an area where the discs are far apart.

What relationship exists between the discs? We want each disc to exert some force on its neighbor. When those discs are far apart, we want the force between them to pull the discs together. On the other hand, when the discs are close together, we want the force between them to push them apart. A simple, physically reasonable force with these properties is a spring

Each spring has a natural length, which is the length of the spring when no forces are being exerted by the spring. *Hooke's Law* for springs says that a stretched spring exerts a restoring force proportional to the distance that the

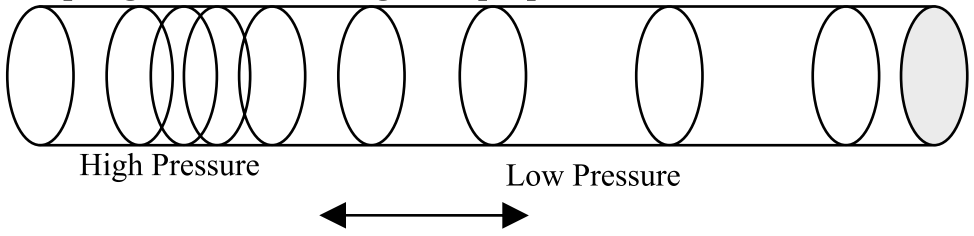


Figure 5: High pressure and low pressure in the disc model

spring is stretched from its natural length. If the natural length of a spring is h and its current length is x, then Hooke's Law implies that the force F exerted by the spring is

$$F = -\tau(x-h)$$

where τ a some positive constant, called the spring constant.

We suppose that the force between the discs in our rod acts like a small spring.

Label the discs 0, 1, 2, ..., n, starting from the left end of the rod. All of the discs and springs in our rod are identical, so we assume that the spring constant τ is the same for every spring. Further, if the total length of the rod is L, the natural length of each spring is

$$h = L/n. \tag{1}$$

Let the distance from the end of the rod to disc i be denoted by y_i

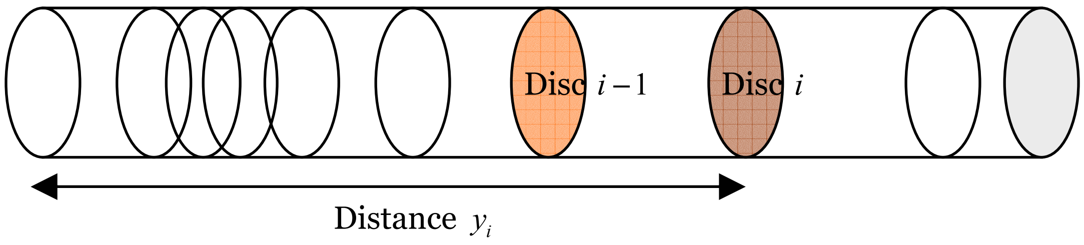


Figure 6: Distances in the disc model

We want to determine the force acting on disc i. First we shall compute the force exerted by disc $i-1$. The distance from disc $i-1$ to disc i is $y_i - y_{i-1}$. Thus, Hooke's Law says that the force $F_{i,i-1}$ exerted on disk i by disc $i-1$ is

$$F_{i,i-1} = -\tau\left(y_i - y_{i-1} - h\right).$$

Similarly the force exerted by disc $i+1$ on disc i is

$$F_{i+1,i} = -\tau\left(y_{i+1} - y_i - h\right).$$

The force exerted by disc i on disc $i+1$ is equal and opposite to the force exerted by disc $i+1$ on disc i; consequently, the force exerted by disc i on disc $i+1$ is

$$F_{i,i+1} = \tau\left(y_{i+1} - y_i - h\right).$$

Since these are all of the forces acting on disc i, we conclude that the total force acting on disc i is

$$\begin{aligned} F_i &= \tau\left(y_{i+1} - y_i - h\right) - \tau\left(y_i - y_{i-1} - h\right) \\ &= \tau\left(y_{i+1} - 2y_i + y_{i-1}\right). \end{aligned}$$

Now that we know the force acting on disc i, we can determine the motion of this disc with the aid of Newton's Law. In particular, since the acceleration of disc i is y_i'' we know that

$$my_i'' = \tau\left(y_{i+1} - 2y_i + y_{i-1}\right)$$

where m is the mass of disc. Thus

$$y_i'' = \frac{\tau}{m}\left(y_{i+1} - 2y_i + y_{i-1}\right).$$

Note that this equation only holds for discs $i = 1, 2, \ldots, n - 1$. It does not hold for disc 0 or for disc n because they do not have two adjacent discs.

To finish our model, we need to determine the behavior of the end discs. Since we do not want the rod to change its total length, we shall fix the end discs at the endpoints of the rod. In particular, we require

$$\begin{aligned} y_0(t) &= 0, \\ y_n(t) &= L \end{aligned}$$

for all values of t.

Our model for the spread of a compression wave is then

$$\begin{aligned} &y_i'' = \frac{\tau}{m}\left(y_{i+1} - 2y_i + y_{i-1}\right) \quad 1 \le i \le n-1, \\ &y_0 = 0, \\ &y_n = L. \end{aligned} \tag{2}$$

The problem with this model is that the parameters m and τ are not known. For this model to be useful, we need to relate these to parameters that we can determine.

Since there are a total of $n+1$ discs, each of mass m, then we know that the total mass of the rod is $(n+1)m$, and thus the density ρ of the rod is

$$\rho = \frac{(n+1)m}{AL}$$

where the cross sectional area of the rod is A. Because the rod is split into a large number of discs, we know n is large, and hence

$$\frac{n+1}{n} \approx 1.$$

Thus if we use (1), we see that

$$\rho = \frac{(n+1)m}{AL} = \frac{n+1}{n}\frac{1}{L/n}\frac{m}{A} \approx \frac{m}{hA}$$

and hence in our equation we can replace m by $hA\rho$.

The *Young's Modulus* of a solid, denoted by E, is defined to be the ratio

$$E = -\frac{F/A}{\delta\ell/\ell}$$

where F is an applied force, A is the cross sectional area, ℓ is the length of a portion of the solid, and $\delta\ell$ is the change in the length of that portion caused by the force. Young's modulus measures the compressibility of a solid.

Since we model the force between adjacent discs by springs with natural length *h*, we know that $F = -\tau \cdot \delta\ell$ and $\ell = h$ so that

$$E = \frac{\tau \cdot \delta\ell}{A}\frac{h}{\delta\ell} = \frac{\tau h}{A}.$$

Thus, we can replace τ by *EA/h*.

Combining these facts, we can rewrite the first equation of our model (2) as

$$y_i'' = \frac{EA/h}{hA\rho}\left(y_{i+1} - 2y_i + y_{i-1}\right)$$

leaving us with the model

$$\begin{aligned} y_i'' &= \left(\frac{E}{\rho}\right)\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \quad 1 \le i \le n-1, \\ y_0 &= 0, \\ y_n &= L. \end{aligned} \tag{3}$$

For a particular solid, the values of *E* and ρ can be determined by consulting a table of physical constants. For instance, for copper, $E = 130 \times 10^9$ kg/m s^2 and $\rho = 8933$ kg/m^3.

Section 3: Numerical Methods for the Compression Wave Model

To solve this problem, let us choose a time step k, and define the time steps $t_j = jk$, which is the time j time steps after $t = 0$. Denote the position of disc i at time t_j by $y_i\left(t_j\right) = y_{i,j}$.

Taylor's Theorem tells us that

$$y_i\left(t_j + k\right) = y_i\left(t_j\right) + y_i'\left(t_j\right)k + \tfrac{1}{2}y_i''\left(t_j\right)k^2 + \tfrac{1}{6}y_i'''\left(t_j\right)k^3 + \tfrac{1}{24}y_i^{(4)}\left(t_j\right)k^4 + O\left(k^5\right)$$

while

$$y_i\left(t_j - k\right) = y_i\left(t_j\right) - y_i'\left(t_j\right)k + \tfrac{1}{2}y_i''\left(t_j\right)k^2 - \tfrac{1}{6}y_i'''\left(t_j\right)k^3 + \tfrac{1}{24}y_i^{(4)}\left(t_j\right)k^4 + O\left(k^5\right).$$

Thus, if we add these and simplify, we see that

$$y_i''\left(t_j\right) = \frac{y_i\left(t_{j+1}\right) - 2y_i\left(t_j\right) + y_i\left(t_{j-1}\right)}{k^2} + O\left(k^2\right)$$

or

$$y_i''(t_j) = \frac{y_{i,j+1} - 2y_{i,j} + y_{i,j-1}}{k^2} + O(k^2).$$

Substituting this into (3), we obtain

$$\frac{y_{i,j+1} - 2y_{i,j} + y_{i,j-1}}{k^2} \approx y_i''(t_j) = \left(\frac{E}{\rho}\right)\frac{y_{i+1,j} - 2y_{i,j} + y_{i-1,j}}{h^2}.$$

Thus if we set

$$\lambda = \frac{E}{\rho}\left(\frac{k}{h}\right)^2$$

we obtain the equation

$$y_{i,j+1} - 2y_{i,j} + y_{i,j-1} = \lambda\left(y_{i+1,j} - 2y_{i,j} + y_{i-1,j}\right). \tag{4}$$

If we know the values of y_i at some time t_j and at the prior time t_{j-1}, then we can use (4) to determine the values of y_i at the next time t_{j+1}.
Indeed if $1 \le i \le n-1$ then

$$y_{i,j+1} = \lambda y_{i+1,j} + 2(1-\lambda)y_{i,j} + \lambda y_{i-1,j} - y_{i,j-1}.$$

while the boundary conditions in (3) yield

$$y_{0,j+1}(t) = 0,$$
$$y_{n,j+1}(t) = L.$$

To use this algorithm, we need to know the position of the discs at time $t_0 = 0$ and at time $t_1 = k$, because with this information we can calculate the position of the discs at time t_2 and at subsequent times. We can require that the user specify the initial position of all of the discs, giving us the position of the discs at time t_0, but how can we determine the position of the discs at time t_1? It would be unnatural to specify these values, because we want to be able to modify the value of k.

We require the user also specify the *velocity* of all of the discs at time $t_0 = 0$. In particular, let v_i be the velocity of disc i at time $t_0 = 0$. We shall then use this information to determine the position of the discs at time t_1. The algorithm then determines the position of the discs at all later times.

To find the position of disc i at time $t_1 = k$, we shall apply Taylor's Theorem. Thus

$$y_i(k) = y_i(0) + ky_i'(0) + \tfrac{1}{2}k^2 y_i''(0) + O(k^3).$$

On the other hand, equation (3) at $t = 0$ tells us that

$$y_i''(0) = \left(\frac{E}{\rho}\right)\frac{y_{i+1,0} - 2y_{i,0} + y_{i-1,0}}{h^2}.$$

Combining these, we then find that

$$y_{i,1} = y_{i,0} + kv_i + \frac{1}{2}\frac{E}{\rho}\left(\frac{k}{h}\right)^2 \left(y_{i+1,0} - 2y_{i,0} + y_{i-1,0}\right) + O\left(k^3\right).$$

Thus, knowing the positions $y_{i,0}$ and the velocities v_i at time $t_0 = 0$, we find the position at time $t_1 = k$ for $1 \le i \le n-1$ by

$$y_{i,1} = \frac{\lambda}{2} y_{i+1,0} + \left(1-\lambda\right) y_{i,0} + \frac{\lambda}{2} y_{i-1,0} + kv_i. \tag{5}$$

Section 4: Partial Derivatives

Our model for the compression wave is

$$y_i'' = \left(\frac{E}{\rho}\right)\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}.$$

and we approximated the left side by

$$y_i''\left(t_j\right) \approx \frac{y_i\left(t_{j+1}\right) - 2y_i\left(t_j\right) + y_i\left(t_{j-1}\right)}{k^2}$$

so that

$$\frac{y_{i,j+1} - 2y_{i,j} + y_{i,j-1}}{k^2} \approx y_i''\left(t_j\right) = \left(\frac{E}{\rho}\right)\frac{y_{i+1,j} - 2y_{i,j} + y_{i-1,j}}{h^2}.$$

The similarity between the left side and right side makes one think that there is some relationship between them.

Let $y\left(x,t\right)$ be a function of the two variables x and t. If we were to treat x as a constant, then the result is a function of one variable $t \mapsto y\left(x,t\right)$. Because this is a function of one variable, we can take its derivative. We call the result the *partial derivative of y with respect to t*, and denote it by either y_t or $\frac{\partial y}{\partial t}$. Similarly, if we hold t fixed and consider the result a function of the one variable x, we can differentiate the result to obtain *partial derivative of y with respect to x*, which we denote y_x or $\frac{\partial y}{\partial x}$.

In the previous section we proved that

$$y_i''\left(t_j\right) = \frac{y_i\left(t_{j+1}\right) - 2y_i\left(t_j\right) + y_i\left(t_{j-1}\right)}{k^2} + O\left(k^2\right),$$

so in the language of partial derivatives we have shown that

$$y_{tt}\left(x_i, t_j\right) = \frac{y_{i,j+1} - 2y_{i,j} + y_{i,j-1}}{k^2} + O\left(k^2\right).$$

On the other hand, if we fix t and let x be the variable, Taylor's theorem says

$$y\left(x_i+h,t_j\right)=y\left(x_i,t_j\right)+y_x\left(x_i,t_j\right)h+\tfrac{1}{2}y_{xx}\left(x_i,t_j\right)h^2+\tfrac{1}{6}y_{xxx}\left(x_i,t_j\right)h^3+\tfrac{1}{24}y_{xxxx}\left(x_i,t_j\right)+O\left(h^5\right)$$

while

$$y\left(x_i-h,t_j\right)=y\left(x_i,t_j\right)-y_x\left(x_i,t_j\right)h+\tfrac{1}{2}y_{xx}\left(x_i,t_j\right)h^2-\tfrac{1}{6}y_{xxx}\left(x_i,t_j\right)h^3+\tfrac{1}{24}y_{xxxx}\left(x_i,t_j\right)+O\left(h^5\right)$$

Adding, we see that

$$\frac{y_{i+1,j}-2y_{i,j}+y_{i-1,j}}{h^2}=y_{xx}\left(x_i,t_j\right)+O\left(h^2\right)$$

Thus our model appears to be an approximation of the equation

$$y_{tt}=\left(\frac{E}{\rho}\right)y_{xx}.$$

This is a *partial differential equation*, called the *wave equation*. Our finite difference method that we have learned is a way to approximate solutions of the wave equation.

Section 5: Transverse Waves

Now that we have examined the behavior of compression waves, let us discuss the behavior of transverse waves. In particular, let us examine a vibrating string.

To model this, begin by assuming that the position of the string can be modeled by a function $u(x,t)$, which describes the height of the string at position x and at time t.

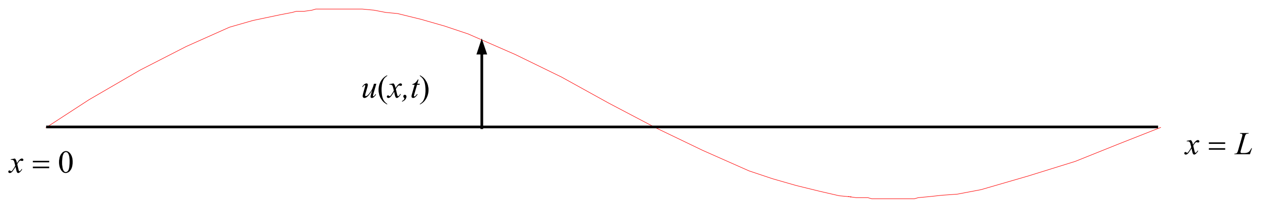


Figure 7: A vibrating string

Examine the portion of the string between two points, say $x=a$, and $x=b$. The forces acting on this portion of the string are the tension at $x=a$ and the tension at $x=b$. Let $T(x,t)$ be the tension in the string at a point x at time t. Because the tension is the force exerted by the rest of the string, it must act in the direction that is tangent to the position of the string at that point.

To find the tension at the point $x=b$, let α be the angle between the tangent line to $u(x,t)$ at $x=b$ and a horizontal line. The slope of $u(x,t)$ at time t and position $x=b$ is $u_x(b,t)$. Thus, to find the angle α, we draw a right triangle whose hypotenuse has slope $u_x(b,t)$, and we see that $\sin\alpha = \frac{u_x}{\sqrt{1+u_x^2}}$ and $\cos\alpha = \frac{1}{\sqrt{1+u_x^2}}$.

Combining these facts, we find that the horizontal component of the tension T at $x = b$ is

$$\frac{T(b,t)}{\sqrt{1+u_x^2(b,t)}}$$

while the vertical component is

$$T(b,t)\frac{u_x(b,t)}{\sqrt{1+u_x^2(b,t)}}.$$

Repeating the previous process at $x = a$, we find that the total horizontal force on the portion of the string between $x = a$ and $x = b$ is

$$\frac{T(b,t)}{\sqrt{1+u_x^2(b,t)}} - \frac{T(a,t)}{\sqrt{1+u_x^2(a,t)}}.$$

If we assume that the string moves only up and down, we know that there can be no horizontal force on the portion of the string between $x = a$ and $x = b$; thus

$$\frac{T(b,t)}{\sqrt{1+u_x^2(b,t)}} - \frac{T(a,t)}{\sqrt{1+u_x^2(a,t)}} = 0.$$

We conclude that there is a value T_0 so that

$$\frac{T(x,t)}{\sqrt{1+u_x^2(x,t)}} = T_0$$

for all values of *x*, for each time *t*. For simplicity, we also assume that T_0 does not change with time, and hence is constant.

The vertical component of the tension force on the portion of the string between $x = a$ and $x = b$ is

$$T(b,t)\frac{u_x(b,t)}{\sqrt{1+u_x^2(b,t)}} - T(a,t)\frac{u_x(a,t)}{\sqrt{1+u_x^2(a,t)}}$$

which can be simplified to

$$T_0\left[u_x(b,t) - u_x(a,t)\right].$$

We shall rewrite this with the aid of the Fundamental Theorem of Calculus as

$$T_0 \int_a^b u_{xx}(x,t) \; dx.$$

We know from Newton's law that $F = ma$. Since we know the force on the portion of the string between $x = a$ and $x = b$, let us try to determine the acceleration of that portion of the string.

We immediately see a problem- different portions of the string are moving at different velocities, and have different accelerations. However, suppose we take some infinitesimally small part of the string, of width dx. On this portion of the string, the acceleration of the string is $u_{tt}(x,t)$, and the mass is $\rho\, dx$, where ρ is the density of the string per unit length. Thus, for this infinitesimally small part, we can write ma as $\rho\, u_{tt}(x,t)\, dx$. Adding up the contributions for each inifintesimal piece between $x = a$ and $x = b$, we obtain

$$\int_a^b \rho u_{tt}(x,t) \; dx.$$

Combining these results, we see that for every choice of *a* and *b*,

$$T_0 \int_a^b u_{xx}(x,t) \; dx = \int_a^b \rho u_{tt}(x,t) \; dx.$$

Thus

$$\int_a^b \left[\rho u_{tt}(x,t) \; - T_0 u_{xx}(x,t)\right] \; dx = 0.$$

This says that the area under $\rho u_{tt}(x,t) \; - T_0 u_{xx}(x,t)$ between any two points $x = a$ and $x = b$ is zero. The only way that this can occur is if the function $\rho u_{tt} - T_0 u_{xx}$ is always zero.

As a consequence, we conclude that

$$u_{tt} = \left(\frac{T_0}{\rho}\right) u_{xx}.$$

Note that, aside from the constant factors, this is the same equation as the equation for a compression wave,

$$y_{tt} = \left(\frac{E}{\rho}\right) y_{xx}.$$

For this reason, equations of the form

$$y_{tt} = c y_{xx}.$$

for $c > 0$ are called a *wave equations*.

Section 6: Speed of Propagation and the Stability of the Numerical Method

The wave equation has an interesting property. Let $f(x)$ be any twice differentiable function. Then the functions

$$y(x,t) = f\left(x - \sqrt{c}t\right)$$

and

$$y(x,t) = f\left(x - \sqrt{c}t\right)$$

both satisfy the wave equation; this can be verified by direct substitution. For each time t, these solutions are translations of the function $y(x,0)$. Moreover, at time t, these solutions have been translated by a distance $\sqrt{c}t$. Thus, the *velocity* of these solutions is $\sqrt{c}t/t = \sqrt{c}$. These are called *traveling wave* solutions of the wave equation.

The fact that traveling wave solutions of the wave equation move at the particular velocity $\sqrt{c}$ has a dramatic effect on the choice of the parameters in our numerical method. Indeed let the spatial grid size h and the time step k be chosen. The value of the solution at $\left(x_i, t_{j+1}\right)$ depends on the values of the solution at $\left(x_{i-1}, t_j\right)$, $\left(x_i, t_j\right)$, and $\left(x_{i+1}, t_j\right)$. Thus, the velocity of the numerical solution is at most $\Delta x/\Delta t = h/k$. If the numerical method is to be accurate, then it needs to move information between the grid points at least as quickly as the actual solution. This translates to the requirement that $\frac{h}{k} \geq \sqrt{c}$, or equivalently the restriction

$$k \leq \sqrt{c}h\,.$$

Although the argument underlying this requirement can be challenged, the result is true.

Note to the Instructor

One could replace the heuristic argument of Section 6 with a rigorous argument based on von Neumann analysis. See the chapter on diffusion problems for the relevant details.

Project

Write a C++ program that simulates the motion of a vibrating string.

As input, the program should take

- The values of the solution at the endpoints

- The ratio of tension to density T_0/ρ
- The user should be able to use the mouse to determine the initial data.
- The step size for the simulation.
- The total time for the simulation.

As output, the program should return

- An animation of the resulting motion of the string.

The program should be written using good object oriented programming techniques.

You are then to answer the following questions

1. Consider a string of length $L = 100\,\text{cm}$ with $T_0/\rho = 10\,\text{N}\cdot\text{m}/\text{kg}$. Suppose that the string is initially at rest, the endpoints are held fixed, but initially is stretched as follows.

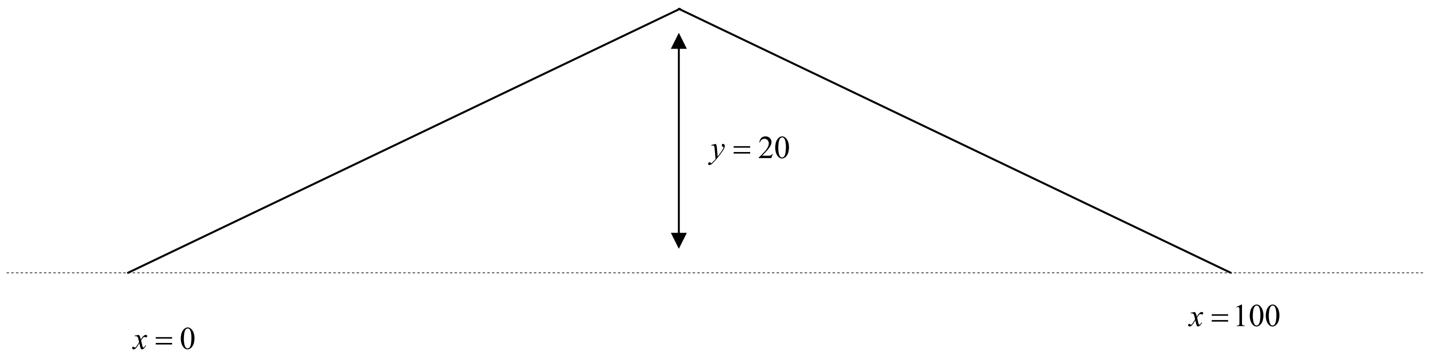


Figure 8: A stretched string

- What is the position of the string when $t = 1$?
- Describe in words the motion of the string.

2. Consider a string of length $L = 100\,\text{cm}$ with $T_0/\rho = 10\,\text{N}\cdot\text{m}/\text{kg}$. Suppose initially that $y(x,0) = y_t(x,0) = 0$, but that the left end oscillates so that $y(0,t) = \sin t$ and the right end is held fixed.

- What is the position of the string when $t = 1$?
- Describe in words the motion of the string.