7

Introduction to Maple Programming

Solvents and Solutes

Chemical engineers apply principles of chemistry, physics, and engineering to the design and operation of industrial plants for the production of materials that are mixed and undergo other chemical changes during the manufacturing process. Reactors and reactor models can be used to synthesize and analyze the components that are combined to create products such as shampoos, cosmetics, plastics and drugs. The reactor model developed in this chapter's application is based on the principle of *conservation of mass*. The use of Maple in the analysis of this model involves a user-defined procedure written in the Maple programming language. In addition to introducing the Maple programming language, you will find discussions on the use of conditional and looping statements.

INTRODUCTION

The first six chapters presented techniques for direct interaction with Maple: enter a command, receive a response. While this is sufficient for many situations, there are times when repeatedly stepping through complicated multi-command sequences is inconvenient and inefficient. The main topic of this chapter is the use of Maple as a programming language to create user-defined commands, including loops, conditionals, input arguments, error handling, and return values.

The simplicity of Maple programs, more properly called *procedures*, is derived from the fact that the programming is done using standard Maple commands. The power of the Maple programming language is evidenced by the fact that almost all Maple commands are implemented in the Maple programming language. Moreover, as you will learn in Section 7.1, the definitions can be viewed, and even modified, by the user.

Some of the examples and problems present a final look at examples and applications from previous chapters. The application introduced in this chapter investigates some of the analytical techniques used by chemical engineers in the design of reactors for the mixing of solvents and solutes.

7-1 VIEWING MAPLE PROCEDURES

Almost all Maple commands are written in Maple, and these definitions can be viewed by the user. Thus, a natural way to begin to learn about the Maple programming language is by looking at the way these commands are implemented. The **showstat** command (which is new in Release 4) displays a procedure in a format that clearly illustrates the structure of the procedure specified in the first argument.

EXAMPLE 7-1 Viewing a Maple Procedure

Use **showstat** to display the definition of the **conjugate** function.

SOLUTION

```
> showstat( conjugate );

conjugate := proc(a)
   1    if nargs <> 1 then
   2      ERROR(`expecting 1 argument, got `.nargs)
        elif type(a,`complex(numeric)`) then
   3      subs((-1)^(1/2) = -(-1)^(1/2),a)
        elif type(a,anything^integer) then
   4      conjugate(op(1,a))^op(2,a)
        elif type(a,`*`) and type(op(1,a),numeric) then
   5      op(1,a)*conjugate(subsop(1 = 1,a))
        else
   6      `conjugate/conjugate`(a)
        fi
end
```

Note that the first unnumbered line of the definition says that **conjugate** is defined to be a *procedure* (**proc** ... **end**) with one formal argument (**a**). The body of the procedure, the lines numbered 1 through 6, illustrates Maple's *conditional statement* (**if** ... **then** ... **elif** ... **else** ... **fi**). The **end** indicates the end of the procedure's body. The value returned by **conjugate** is the value of the last command executed in the procedure.

The conditional statement is not difficult to understand. The Boolean expression in line 1 checks the arguments for errors. If **conjugate** is called with more than one argument, or no arguments, then **nargs<>1** will evaluate to **true**, execution of **conjugate** will terminate, and an error message will be generated. If there is exactly one argument, then processing continues by checking the Boolean expressions in the **elif** lines. If all of the Boolean expressions evaluate to **false**, then the command following the **else** is executed.

Try It

Determine input arguments that will be processed by each of the five different cases in the definition of **conjugate**.

Since a procedure is a Maple object, the **print** command provides a second method for viewing the definition of a command. One complication with the use of **print** is that, by default, Maple does not display the body of procedures. To override this default, the command **interface(verboseproc=2):** should be executed. (The **eval** command produces the same output as **print**.)

EXAMPLE 7-2

Another Method for Viewing a Maple Procedure

Use **print** to display the definition of **conjugate**. How does this compare with the display produced by **showstat**?

SOLUTION

Before the interface default is changed, printing a procedure simply shows that **conjugate** is a procedure with one formal parameter.

```
> print( conjugate );

proc(a) ... end
```

When the printing of procedure bodies is enabled, the result is

```
> interface( verboseproc=2 ):
> print( conjugate );

proc(a)
option `Copyright (c) 1995 by Waterloo Maple Inc. All rights reserved.`;
 if nargs ≠ 1 then ERROR (`expecting 1 argument, got `.nargs)
 elif type (a, `complex(numeric)`) then subs(I=-I,a)
 elif type (a, anything^integer) then conjugate(op(1,a))^op(2,a)
 elif type (a, `*`) and type(op(1,a), numeric) then
  op(1,a)*conjugate(subsop(1=1,a))
 else `conjugate/conjugate`(a)
 fi
end
```

Differences between the output produced by **print** and by **showstat** include the inclusion of line numbers by **showstat** and the use of different fonts and styles to distinguish names, functions, and reserved words by **print**. Note also that **I** is displayed by **print**, but **showstat** uses the internal representation **((-1)^(1/2))**.

For short procedures, the differences between **print** and **showstat** are often a matter of personal taste. For longer procedures, however, the ability to display selected lines from a procedure with **showstat** can be extremely useful. The line numbers correspond to the line numbers produced by **showstat** and are specified in the optional second argument as a single number or as a range.

EXAMPLE 7-3

Viewing a Portion of a Procedure

Display the first three lines of the definition of **conjugate**.

SOLUTION

```
> showstat( conjugate, 1..3 );

conjugate := proc(a)
   1    if nargs <> 1 then
   2      ERROR(`expecting 1 argument, got `.nargs)
        elif type(a,'complex(numeric)') then
   3      subs((-1)^(1/2) = -(-1)^(1/2),a)
        elif type(a,anything^integer) then
          ...
        elif type(a,`*`) and type(op(1,a),numeric) then
          ...
        else
          ...
        fi
end
```

The **proc** and **end** lines are always displayed by **showstat**. The conditional command (line 1) includes the **elif**, **else**, and **fi** lines even though they do not appear within the specified lines in the definition.

Try It

The definition of the **dsolve** command is quite long (93 numbered lines). The first few lines check the input arguments, then a conditional is used to determine the method to use to attempt to solve the equation. Use **showstat** to a) display the part of **dsolve** that looks for errors in the arguments and b) display the conditional statement that shows the tests Maple applies to determine the algorithm to use to attempt to solve the equation.

The only commands that cannot be displayed using **showstat**, **print**, or **eval** are the *built-in commands*. These are typically the most fundamental commands and other commands whose Maple implementation would be noticeably inefficient. Examples of built-in commands include **subs**, **op**, **sort**, and **diff**.

7-2 FUNCTIONAL OPERATORS

As the examples in Section 7.1 illustrate, the **proc ... end** command is the primary Maple command used to define a Maple procedure. The arrow operator (**->**), which you first encountered in Chapter 3, can be used to define simple procedures, called *functional operators*, which consist of a single command. In general, while the arrow operator is somewhat simpler to use than **proc ... end**, its main use is for implementing mathematical functions, for example, **f := (x,y) -> (x^2-y^2)/(x^2+y^2);**.

EXAMPLE 7-4 Example 3-8 Revisited with Functional Operators

In Example 3-8 you created the sorted list of integers, through 1 million, that are both perfect squares and cubes. Use the arrow operator and the **member** command to define a procedure, **checkSC**, whose input argument is a single integer, *n*, and returns true if *n* is both a perfect square and cube and false in all other cases.

SOLUTION

From the solution to Example 3-8, the following commands create the sorted list of perfect squares and cubes that do not exceed 1 million.

```
> SQR := { i^2 $ i=1..1000 }:
> CUB := { i^3 $ i=1..100 }:
> SClist := sort( convert( SQR intersect CUB, list ) ):
```

The simplest Maple procedure that meets the requirements of this problem is

```
> checkSC := n -> member( n, SClist );
```

$$checkSC := n \to \mathrm{member}(n, SClist)$$

To test this procedure, call **checkSC** for a variety of input arguments.

```
> checkSC( 64 );
```

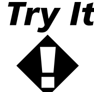
true

```
> checkSC( 65 );
```

false

The implementation of **checkSC** in Example 7-4 assumes that the argument is a single number. One means of overcoming this limitation is the **map** command, which applies a procedure, specified as its first argument, to each operand of the expression specified in the second argument. If the procedure requires additional arguments, these can be specified as optional arguments to **map**.

Try It



Determine, in a single statement, which powers of 2 are in **SClist**. (*Hint*: Use $ to create the list of all powers of 2 that do not exceed 1 million.)

EXAMPLE 7-5

A Recursive Extension to Example 7-4

Implement, using the arrow operator and recursion, a version of **checkSC** that is automatically applied to each element of any set or list that is specified as an argument.

SOLUTION

One way to approach this problem is with *recursion*. In this case, this means that the basic definition is to be applied only when the argument contains a single operand; otherwise, the function should be mapped to each operand. For example,

```
> checkSC2 := n -> if nops(n)=1
>                      then member( n, SClist )
>                      else map( checkSC2, n )
>                  fi;

checkSC2 := proc(n)
option operator, arrow;
    if nops(n) = 1 then member(n, SClist)
    else map(checkSC2, n)
    fi
end
```

Testing of **checkSC2** should include examples that can be evaluated with and without recursion.

```
> checkSC2( 64 );
```

true

```
> checkSC2( 65 );
```

false

```
> checkSC2( [ k^3 $ k=1..20 ] );
```

[*true*, *false*, *false*, *true*, *false*, *false*, *false*, *false*, *true*, *false*, *false*, *false*, *false*, *false*, *false*, *true*, *false*, *false*, *false*, *false*]

Try It

It might seem that the simpler definition **checkSC3 := n -> map(member,n,SClist);** fulfills the requirements of Example 7-5, without resorting to recursion. Show that **checkSC2** and **checkSC3** are not equivalent by finding one argument for which different results are obtained.

Recursion can be very useful, but it should not be overused. In particular, one of the quickest ways to kill a Maple session is to define a recursive function whose stopping state is never satisfied. (See the *Maple V Programming Guide*, Section 1.2, for additional comments on the use of recursion in Maple.)

EXAMPLE 7-6

A Purely Procedural Solution to Example 7-4

The various implementations of **checkSC** all depend on the prior existence of **SClist**, the list of perfect squares and cubes through 1 million. Create a functional operator **testSC** that can be applied for any integer argument, or set or list of integers. Test **testSC** on the list **[32^k $ k = 0 .. 12]**.

SOLUTION

If the collection of perfect squares and cubes is not computed in advance, then it will be necessary to implement a test that determines whether an integer is a perfect square and a perfect cube. One possibility is

```
> testSC := n -> if nops(n) = 1
>                 then evalb( ispower(n,2) and ispower(n,3) )
>                 else map( testSC, n )
>               fi;

testSC := proc(n)
option operator, arrow;
    if nops(n) = 1 then evalb(ispower(n, 2) and ispower(n, 3))
    else map(testSC, n)
    fi
end
```

where **ispower** is a Maple function, which has not yet been written, that accepts two integer arguments (n,p), with $p > 0$, and returns **true** if $n = m^p$ for some integer m and **false** otherwise. Since $n = m^p$ for some integer m if and only if $n^{1/p}$ is an integer, this suggests the use of **surd** to implement **ispower**:

```
> ispower := ( NUMBER, POWER ) -> type( surd( NUMBER, POWER ), integer );

ispower := (NUMBER, POWER) -> type(surd(NUMBER, POWER), integer)
```

To conclude, you should first test that **ispower** is working correctly

```
> ispower( 32, 2 );
```

$$false$$

```
> ispower( 32, 5 );
```

$$true$$

Since these results are correct, you should conclude by conducting trials with **testSC**:

```
> testSC( [ 32, 64 ] );
```

[*false*, *true*]

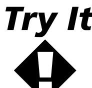
```
> pow32 := [ 32^k $ k=0..12 ];
```

pow32 := [1, 32, 1024, 32768, 1048576, 33554432, 1073741824, 34359738368, 1099511627776, 35184372088832, 1125899906842624, 36028797018963968, 1152921504606846976]

```
> testSC( pow32 );
```

[*true*, *false*, *false*, *false*, *false*, *false*, *true*, *false*, *false*, *false*, *false*, *false*, *true*]

Thus, from this list of 13 integers, only 3 are perfect squares and cubes.

Try It 

Compare the results obtained in Example 7-6 with those obtained by applying the appropriate version of **checkSC** to **pow32**.

The specific integers from **pow32** that are perfect squares and cubes can be found by careful counting. The **select** command provides a more general method of selecting elements of a list, set, sum, product, or function that meet a prescribed Boolean condition. The **remove** command is similar, except that the output is formed by removing all elements that meet the condition. For example, to extract the five entries from **pow32** that are perfect cubes

```
> select( ispower, pow32, 3 );
```

[1, 32768, 1073741824, 35184372088832, 1152921504606846976]

Try It

Use **select** and **testSC** to identify the three elements of **pow32** that were identified as perfect squares and cubes in Example 7-6.

The **map** command, and its close relative **map2**, can be used for functions with more than one argument. For example, **map(ispower, pow32, 3);** checks if each element of the **pow32** is an integer power of 3. Similarly, **map2(ispower, 64, [2, 3, 4]);** determines if 64 is a perfect square, cube, or quartic power. Note that, for both **map** and **map2**, only one argument of the function changes and the difference between **map** and **map2** is the location of the argument that changes.

7-3 TYPE CHECKING, ERROR, AND RETURN

The **testSC** procedure defined in Example 7-6 is written with the implicit assumption that the argument is an integer or a compound object whose operands are integers.

EXAMPLE 7-7

Using testSC with Non-integer Arguments

Consider the implementation of **testSC** and **ispower** from Example 7-6. What response do you expect from the command **testSC(1/64);**? What response is produced by this command?

SOLUTION

From the context of the original definition of **checkSC**, it would seem that this command should produce either an error message or **false**. In fact, execution of this command yields

```
> testSC( 1/64 );

Error, (in testSC) too many levels of recursion
```

The infinite loop is caused by the fact that **nops(1/64)** evaluates to 2, but **map** sees only a single element to which **testSC** should be applied. The same error message is obtained for any rational number.

. . . .

This problem can be avoided if you include explicit checks that the arguments to a user-defined procedure have the correct type and structure. One way to accomplish this is with conditionals involving the **type** command and messages created by the **ERROR** command.

EXAMPLE 7-8

Explicit Type Checking

Modify the definition of **testSC** to include a check that the argument is an integer, set of integers, or list of integers.

SOLUTION

Note that if the arrow operator is to be used, it is necessary to include the entire type checking and definition in a single (conditional) statement. One possibility is

```
> testSC := n ->
>  if not type(n,{integer,set(integer),list(integer)}) then
>   ERROR(`argument must be an integer or a list or set of integers`)
>  elif nops(n)=1 then
>   evalb( ispower(n,2) and ispower(n,3) )
>  else
>   map( testSC, n )
>  fi;

testSC := proc(n)
option operator, arrow;
  if not type(n, {list(integer), set(integer), integer}) then
    ERROR(`argument must be an integer or a list or set of integers`)
  elif nops(n) = 1 then evalb(ispower(n, 2) and ispower(n, 3))
  else map(testSC, n)
  fi
end
```

Note that normal usage has not been changed

```
> testSC( 64 );
```

true

```
> testSC( [ 1, -1, 64 ] );
```

[*true*, *false*, *true*]

while noninteger data is now reported as invalid

```
> testSC( 1/64 );

Error, (in testSC) argument must be an integer or a list or set of integers

> testSC( [ 1, 2, 1/6 ] );

Error, (in testSC) argument must be an integer or a list or set of integers
```

Observe that even though the arrow operator is used to define **checkSC2** and **testSC**, the printed version that appears immediately after the definition uses the **proc ... end** structure. The only indication that these procedures came from an arrow operator is the line (**option operator, arrow;**).

. . . .

Maple's *type-based pattern matching* provides a second technique to check that input arguments are of a specific type. To use this feature for checking arguments of a procedure, simply append the string **::** and the appropriate type specification immediately following the appropriate name in the argument list, for example **n :: integer**. In particular, **x :: t** is a synonym for **type(x, t)**. Additional information about pattern matching can be found under the help topic **typematch**. For information about the description of types, see the **type**, **type,surface**, and **type,structure** help worksheets.

EXAMPLE 7-9

Automatic Pattern Matching

Rewrite the version of **testSC** in Example 7-8 in a form that uses Maple's type-based pattern matching. Compare the output from this procedure and the one in Example 7-8.

SOLUTION

The only modification that is required is to move the description of a valid argument from the first part of the conditional to the left-hand side of the arrow operator, which must be enclosed in parentheses.

```
> testSC := ( n :: {integer,set(integer),list(integer)} ) ->
>   if nops(n)=1 then
>     evalb( ispower(n,2) and ispower(n,3) )
>   else
>     map( testSC, n )
>   fi;
```

```
testSC := proc(n::{list(integer), set(integer), integer})
option operator, arrow;
    if nops(n) = 1 then evalb(ispower(n, 2) and ispower(n, 3))
    else map(testSC, n)
    fi
end
```

The tests of this implementation show its equivalence to the implementation in Example 7-8:

```
> testSC( 64 );
```

$$true$$

```
> testSC( [ 1, -1, 64 ] );
```

$$[true, false, true]$$

```
> testSC( 1/64 );
```

```
Error, testSC expects its 1st argument, n, to be of type {list(integer),
            set(integer), integer}, but received 1/64
```

```
> testSC( [ 1, 2, 1/6 ] );
```

```
    Error, testSC expects its 1st argument, n, to be of type
{list(integer), set(integer), integer}, but received [1, 2, 1/6]
```

The only difference between the implementations in Example 7-8 and this one is that the error messages created with the automatic pattern matching are more informative than the simple messages generrated in Example 7-8.

Try It

Rewrite the **ispower** procedure so that it generates error messages when the first argument is not an integer and the second argument is not a positive integer. (Select appropriate test cases that test all aspects of the new definition.)

The value returned by a procedure is usually the value of the last evaluated expression before the end of the procedure. One exception to this is the **ERROR** command, which stops execution of the procedure without returning a value and displays the error message that is provided. The **RETURN** command also terminates execution of a procedure, but returns the value of the expression in its argument (after evaluation).

EXAMPLE 7-10

The Fibonacci Sequence

The Fibonacci sequence is the sequence of integers 0, 1, 1, 2, 3, 5, 8, … . The complete mathematical definition of this sequence is $f_n = f_{n-1} + f_{n-2}$ if $n \geq 2$ with $f_0 = 0$ and $f_1 = 1$. Write a Maple procedure that returns the nth Fibonacci number, f_n.

SOLUTION

A Maple procedure, complete with type checking, for the computation of the *n*th Fibonacci number is

```
> fib := proc( n :: nonnegint )
>   if n>=2 then
>      RETURN( fib(n-1)+fib(n-2) )
>   else
>      RETURN( n )
>   fi
> end:
```

Note that neither **RETURN** is necessary in this case, but that the use of **RETURN** in these situations is good programming style. You should develop these habits from the outset.

The first eleven Fibonacci numbers are

```
> [ seq( fib(k), k=0..10 ) ];
```

$$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]$$

The efficient evaluation of a recursively defined function should avoid repeated evaluation of the function for the same argument. In Maple, a *remember table* keeps a record of the input arguments and corresponding return value for all calls to a procedure (see the help worksheet for **remember**). To request that a procedure maintain a remember table, simply specify **option remember;** between the **proc** clause and the first line of the body of the procedure.

EXAMPLE 7-11

Creating a Remember Table

Modify the definition of **fib** in Example 7-10 to create a new procedure, **Fib**, that uses a remember table.

SOLUTION

The only modifications are the renaming of the procedure and the insertion of **option remember**; immediately following the **proc** command.

```
> Fib := proc( n :: nonnegint )
>   option remember;
>   if n>=2 then
>      RETURN( Fib(n-1)+Fib(n-2) )
>   else
>      RETURN( n )
>   fi
> end:
```

A simple test that this function works as expected is:

```
> Fib( 10 );
```

$$55$$

Try It

The number of function calls, c_n, required to compute f_n without the use of a remember table can be shown to satisfy the recurrence relation $c_n = c_{n-1} + c_{n-2} + 1$ for $n \geq 2$ with $c_0 = 1$ and $c_1 = 1$. Write a Maple procedure that efficiently computes c_n.

7-4 REPETITION LOOPS

Most modern programming languages provide some form of loop control structure. In Maple the *repetition statement* is the **for ... while ... do ... od** command. Any combination of the **for** and **while** clauses can be specified; only the **do** and **od** are required.

EXAMPLE 7-12

Using Repetition to Sum a Collection of Numbers

Use the repetition statement to find the sum of the squares of the first ten Fibonacci numbers.

SOLUTION

```
> SUM := 0; for N from 1 to 10 by 1 do SUM := SUM + Fib(N) od;
```

$$SUM := 0$$

$$SUM := 1$$

$$SUM := 2$$

$$SUM := 4$$

$$SUM := 7$$

$$SUM := 12$$

$$SUM := 20$$

$$SUM := 33$$

$$SUM := 54$$

$$SUM := 88$$

$$SUM := 143$$

The syntax of the repetition statement is fairly straightforward. The name following **for** identifies the loop index; the **from** clause sets the initial value for the loop index (if omitted, the initial value is 1); the **by** clause sets the step for each iteration (if omitted, this is also assumed to be 1); the **to** clause identifies the final value of the loop index. If the **to** is omitted, the loop will generally be terminated via a **break** statement inside the repetition loop or a **while** clause. The **while** clause specifies a Boolean condition that is checked at the beginning of each loop (the loop continues as long as this expression evaluates to **true**). The **do** and **od** denote the beginning and end of the sequence of expressions that are executed during each pass through the loop. For a full description of this command please see the help worksheet for **do**.

EXAMPLE 7-13

Another Sum Using Repetition

Use the repetition statement to compute the sum of the square of the first 10 prime numbers.

SOLUTION

A straightforward solution is to start with the smallest prime and add the square of each prime until ten primes have been found. As a cross-check, keep a list of the primes as they are encountered.

To implement this plan it is necessary to keep track of the number of primes that have been found, the list of primes, and the sum of the squares of the primes:

```
> COUNT := 0; PRIMES := NULL; SUM := 0;
```

$$COUNT := 0$$

$$PRIMES :=$$

$$SUM := 0$$

The remainder of the algorithm is represented by

```
> for N from 2 by 1 while COUNT<10 do
>   if isprime(N) then
>      COUNT:=COUNT+1;
>      PRIMES:=PRIMES,N;
>      SUM:=SUM+N^2;
>   fi
> od;
```

In this case, the results of executing this loop are

```
> COUNT; PRIMES; SUM;
```

$$10$$

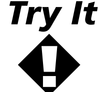
$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

$$2397$$

Moreover, the loop index retains its final value

```
> N;
```

$$30$$

Try It 

Another way to approach Example 7-13 is to use a loop to determine the appropriate collection of primes as a list or set, then use other methods to compute the square of each prime and to find the sum of the squares. Write a Maple procedure, **sumprime**, that computes the sum of the squares of the next n primes greater than or equal to p. Both n and p will normally be specified as arguments; if only one argument is provided, assume it is p and that $n = 1$.

A second variation of the repetition statement is designed for use when the values for the index variable are to be taken from a list or set, not a geometric progression. The syntax for this is the same, except that **for ... in ...** replaces **for ... from ... to ... by ...** .

EXAMPLE 7-14

Using Repetition in a Sum

Write a Maple procedure, **minpos**, that uses the repetition statement to find the smallest positive number from a list or set of numbers. (Be sure to include type checking.) An alternate implementation of **minpos** is discussed in Problem 4 at the end of this chapter.

Use Maple's *random number generator*, **rand**, to generate test data for **minpos**.

SOLUTION

The only real issue here is how to initialize the counter used to keep track of the smallest positive number. Two possibilities are to initialize the counter to the largest element in the set (**MIN := max(DATA)**) or to **NULL** (the empty expression). Using the latter of these options, the procedure can be written as

```
> minpos := proc( DATA :: { set(constant), list(constant) } )
>   MIN := NULL;
>   for N in DATA do
>     if N > 0 then MIN := min( MIN, N ) fi;
>   od;
>   RETURN( MIN );
> end:

Warning, `MIN` is implicitly declared local
Warning, `N` is implicitly declared local
```

The help worksheet for **rand** indicates that a random number generator, **RNG**, with integer values in the interval [–100, 100] is created with the command

```
> RNG := rand( -100 .. 100 );

RNG := proc()
local t;
global _seed;
    _seed := irem(427419669081*_seed, 999999999989);
    t := _seed;
    irem(t, 201) - 100
end
```

Then, a list of 10 random numbers between –100 and 100 is obtained from

```
> data := [ seq( RNG(), k=1..10 ) ];
```

$$data := [-24, -52, 41, 58, 82, 21, 29, -97, -88, 31]$$

```
> minpos( data );
```

21

7-5 LOCAL AND GLOBAL VARIABLES

The definition of **minpos** in Example 7-14 produced a warning message about a variable being "implicitly declared local." Each variable appearing in a Maple procedure is either a local, global, or environment variable. *Local variables* are known only within this call to the procedure and are independent of any other use of this name. *Environment variables* are a special type of global variable (see the help worksheet for the topic **envvar**).

Global variables should be used only when they are absolutely necessary. In situations that justify the use of a global variable, the names of each global variable should be chosen to be descriptive and prevent inadvertant modification by any other Maple procedure. Short names should be avoided unless they are preceded by a special character such as an underscore. While Maple has specific rules for deciding if a variable is local or global by default (see the help worksheet for **procedure**), it is good programming technique to explicitly declare all local and global variables.

Consider the following procedure:

```
> newname := proc()
>   global _N;
>   if not(assigned( _N )) then _N:=0 fi;
>   for _N from _N+1 while assigned( C._N ) do od;
>   RETURN( C._N )
> end:
> _N := '_N':
```

There are two new commands in the definition of **newname**. The **assigned** command is easily understood from the online help worksheet; **.** is the *concatenation* or *dot operator* (see the help topic **dot**), which is used here to append a number to the name **C** to form new Maple names.

EXAMPLE 7-15 Illustrating the Use of Global Variables

Suppose the following assignments are made:

```
> C2:=1: C4:=1.2: C5:=a: C8:=C2:
```

Explain the output produced when the following sequence of commands is executed:

```
> _N;
> newname() * sin(x) + newname() * cos(x);
> _N;
> newname() * exp(x) + newname() * exp(-x);
```

SOLUTION

```
> C2:=1: C4:=1.2: C5:=a: C8:=C2:
```

At the outset, **_N** is unassigned.

```
> _N;
```

$$_N$$

The first call to **newname** assigns **_N** the value 0, then returns **C1** since this name has not been assigned a value. The value of **_N** is now 1. The second time **newname** is called, it takes two steps to find an unassigned name.

```
> newname() * sin(x) + newname() * cos(x);
```

$$C1 \sin(x) + C3 \cos(x)$$

The current value of **_N** is 3.

```
> _N;
```

$$3$$

The third and fourth calls to **newname** are similar. The final value for **_N** should be 7.

```
> newname() * exp(x) + newname() * exp(-x);
```

$$C6\, e^{x} + C7\, e^{(-x)}$$

```
> _N;
```

$$7$$

7-6 DIGGING DEEPER AND DEBUGGING

A complete description of Maple's debugging system cannot be provided within the scope of this module. The commands most commonly used during the development of Maple procedures are **debug**, **trace**, **printlevel**, and **maplemint**. The **debug** and **trace** commands can be used to see the results of each command that is executed in a procedure; once enabled, this feature is disabled with **undebug** or **untrace** (see the help worksheets for **debug**). Each command is associated with a level. By default, the results of all commands at level 1 are displayed; the results of commands at higher levels can be displayed by increasing the value assigned to **printlevel** (see the help worksheet for **printlvl**). The syntax checker, **maplemint**, detects syntax errors and identifies potential problems with local and global variables (see the online help for **maplemint**).

The *Maple debugger* is another tool which can be useful when programming with Maple. With the debugger you can specify breakpoints and watchpoints within a procedure. A *breakpoint* is a line number, as reported by **showstat**, where execution of the procedure pauses; a *watchpoint* interrupts

execution of the procedure whenever a specified variable is assigned a value or an error occurs. For specific information on the use of the Maple debugger, or any of the other programming tools mentioned in this section, consult the online help topics **debugger** and **debug**.

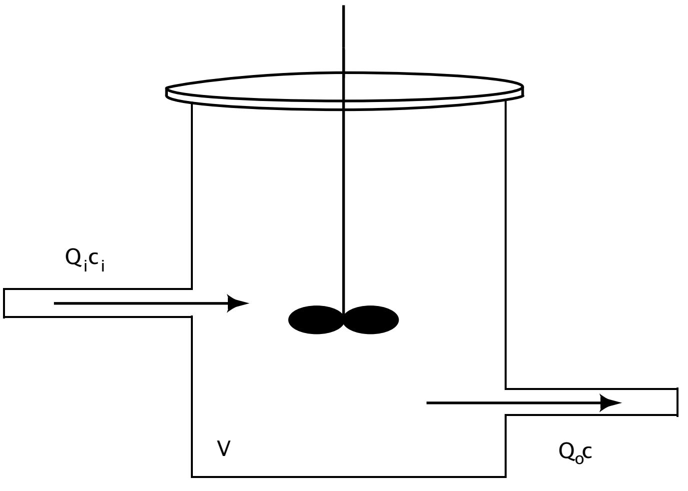
Application 7 SOLVENTS AND SOLUTES

Chemical engineers frequently combine different substances in reactors and employ reactor models to synthesize and analyze a variety of products ranging from drugs to synthetic materials. This application will show how the principle of conservation of mass can be used to develop a model for a simple reactor in which a solute is combined with a solvent to produce a new solution. The model, which leads to a differential equation, will be analyzed graphically and qualitatively without explicitly finding a solution. The two quantities of particular interest in this analysis are the concentration and mass of solvent in the reactor at any time while the reactor is in operation. The amount of usable solution that is produced by this reactor and other issues are discussed in Problems 10–14 at the end of this chapter.

Fundamentals

Figure 7-1 shows a reactor, with volume V (m^3), which is connected to one inflow and one outflow pipe. The mass density of the solution inside the reactor is denoted by ρ (kg/m^3). Thus, the total mass of the solution, which includes both the component solute and the solvent, in the reactor is ρV (kg). The solute component flows into the reactor with a concentration of c_i (mg/m^3) at a rate of Q_i (m^3/min) and the mixed solution is drawn from the reactor at the rate Q_o. Typically, $c_i \ll \rho$. The concentration, c (mg/m^3), of the solute in the outflow pipe is the same as the concentration in the reactor. The reactor is said to be "completely mixed" because the mixing paddle ensures "instantaneous" mixing of the solvent as soon as it is added to the reactor. (Note: This is a modeling approximation that may not be appropriate when the time scale for the mixing is comparable to the flow rates; in such a situation it would be necessary to use a partial differential equation to analyze the spatial variation of the concentration in the reactor model.)

Figure 7-1
Reactor with one inflow and one outflow pipe



1. Define the problem

The principle of conservation of mass states that the total mass of the solute must be constant. Thus, the instantaneous rate of change of the solute mass with respect to time (that is, the accumulation of mass) within the reactor must exactly balance the the difference between the mass entering and leaving the reactor through the two pipes (per unit time). When the tank has a single inflow pipe the (instantaneous) rate at which mass enters the reactor is $Q_i c_i$ (m^3/min × mg/m^3 = mg/min). Similarly, the (instantaneous) rate at which mass is drawn off via the outflow pipe is $M = Q_o c$; the total mass produced during the first t minutes is

$$p(t) = \int_0^t Q_o \, c(\tau)\, d\tau .$$

Compare the concentration and mass during the first 4 hours of operation in two different reactors. Both reactors have an inflow concentration of 50 mg/m^3, an inflow rate of 5 m^3/min, and an initial ($t = 0$) volume of 100 m^3. Both reactors start with the same concentration of solute in the tank; the initial accumulation rate is 200 mg/min for the first reactor. The only difference between the reactors is the outflow rates, which are 5 m^3/min and 5.5 m^3/min, respectively.

2. Gather information

The total mass in the reactor at time t (min) is $M(t) = V(t)c(t)$ ($m^3 \times$ mg/m^3 = mg). Conservation of solute mass is, therefore, expressed as

$$M'(t) = Q_i c_i - \frac{Q_o \, M(t)}{V(t)}$$

or, in terms of the concentration,

$$\frac{d}{dt}\big(V(t)c(t)\big) = Q_i c_i - Q_o \, c(t)$$

Note that, in general, the volume is a function of time. If, as in the first reactor to be analyzed in this problem, the inflow and outflow rates are the same (i.e., $Q_i = Q_o$), then the volume is constant and the differential equations for the solute mass and concentration reduce to

$$M'(t) = Q_i c_i - \frac{Q_i \, M(t)}{V_0} \quad \text{and} \quad M'(t) = Q_i c_i - \frac{Q_o \, M(t)}{V(t)}$$

When the inflow and outflow rates are different, the volume in the reactor is time-dependent. If the flow rates are constant, the volume after t minutes is $V(t) = V_0 + (Q_i - Q_o)t$. The corresponding differential equations will be derived in Step 3.

The last piece of information needed to complete the mathematical description of the reactor is the initial condition. Since only the initial accumulation rate is provided, some work is needed to compute the initial concentration or initial mass. In general, if the initial accumulation rate is A mg/min, that is, $\mathrm{M}'(t) = A$ (at $t = 0$), then, using the differential equation for the mass evaluated at $t = 0$, $A = Q_i c_i - \frac{Q_o \mathrm{M}(0)}{\mathrm{V}(0)}$. From here it is not difficult to obtain an expression for the initial mass in terms of known quantities; dividing the initial mass by the initial volume yields the initial concentration.

The common parameter values for the two reactors are the initial volume, initial accumulation rate, inflow concentration, and inflow rate: $V_0 = 100$ m^3, $A = 200$ mg/min, $Q_i = 5$ m^3/min, and $c_i = 50$ mg/m^3. The first reactor has an outflow rate of $Q_o = 5$ m^3/min while the drain from the second tank flows at a rate of 5.5 m^3/min.

3. Generate and evaluate potential solutions

The problem asks for a qualitiative description of the mass and concentration during a four hour time period. Since you have two different reactors to analyze—and might reasonably expect to be asked to analyze more in the future—a good approach is to create a general procedure for producing plots of the mass and concentration over a given time interval.

The procedure, which will be called **plotCM**, will use **DEplot** to create plots of the mass and concentration and **display** to place the plots side by side. The arguments to **DEplot** must contain the differential equation, the name of the dependent variable (either **M** or **c**), the range of the independent variable (for example, **t = 0 .. 240**), and the initial condition; all other arguments are optional. If conservation of mass is encoded directly into **plotCM**, the arguments to **plotCM** will need to supply the volume (as a function of time), values for all other parameters, and the time interval on which the solutions should be plotted. This suggests that **plotCM** can be divided into three steps: verification of the arguments, derivation of the differential equations and initial conditions, and creation and display of the plots.

The middle step requires the most thought and planning. It can be helpful to interactively test the individual steps involved in the derivation of the initial value problems. To see exactly how each differential equation and initial condition can be obtained from conservation of mass, recall that the principle of conservation of mass can be expressed as

```
> restart;
> with( student ):
> ConsMass := diff( M(t), t ) = Q[i]*c[i] - Q[o]*M(t)/V(t);
```

$$\mathit{ConsMass} := \frac{\partial}{\partial t}\mathrm{M}(t) = Q_i c_i - \frac{Q_o \mathrm{M}(t)}{\mathrm{V}(t)}$$

The specific differential equation for the mass of solute in a reactor is obtained by specifying the volume and values for all other parameters.

```
> odeM := ConsMass;
```

$$odeM := \frac{\partial}{\partial t}\mathrm{M}(t) = Q_i c_i - \frac{Q_o\,\mathrm{M}(t)}{\mathrm{V}(t)}$$

The corresponding differential equation for the concentration can be obtained by using the identity $\mathrm{M}(t) = \mathrm{c}(t)\mathrm{V}(t)$ in the conservation of mass equation and forcing evaluation of the expression, and then solving for the rate of change of the concentration:

```
> odeC := value( subs( M(t) = c(t)*V(t), ConsMass ) );
```

$$odeC := \left(\frac{\partial}{\partial t}\mathrm{c}(t)\right)\mathrm{V}(t) + \mathrm{c}(t)\left(\frac{\partial}{\partial t}\mathrm{V}(t)\right) = Q_i c_i - Q_o\,\mathrm{c}(t)$$

```
> odeC := op( solve( odeC, { diff( c(t), t ) } ) );
```

$$odeC := \frac{\partial}{\partial t}\mathrm{c}(t) = \frac{\mathrm{c}(t)\left(\frac{\partial}{\partial t}\mathrm{V}(t)\right) - Q_i c_i - Q_o\,\mathrm{c}(t)}{\mathrm{V}(t)}$$

The initial conditions for the two differential equations need to be expressed in terms of the initial accumulation rate, A, of the solute, that is, $\mathrm{M}'(t)$ at $\mathrm{t} = 0$. Forcing agreement between the data value and the equation for conservation of mass yields the relation:

```
> InitAccum := A = subs( t=0, rhs( odeM ) );
```

$$InitAccum := A = Q_i c_i - \frac{Q_o\,\mathrm{M}(0)}{\mathrm{V}(0)}$$

Thus, the initial mass, M(0), must be given by

```
> icM := isolate( InitAccum, M(0) );
```

$$icM := \mathrm{M}(0) = \frac{(-A + Q_i c_i)\,\mathrm{V}(0)}{Q_o}$$

and the corresponding initial concentration, c(0), is

```
> icC := isolate( subs( M(0)=c(0)*V(0), icM ), c(0) );
```

$$icC := \mathrm{c}(0) = \frac{-A + Q_i c_i}{Q_o}$$

Recall that the two reactors have different outflow rates but have the same initial conditions. Since all other parameters are the same, the initial accumulation rate must be different for the two reactors. In the current context, it is easier to compute the initial mass and pass this value, along with the values for c_i, Q_i, and Q_o, to **plotCM**. For this reason it seems prudent to implement the initial conditions as

```
> icM := M(0) = M[0];
> icC := c(0) = M[0]/V(0);
```

$$icM := \mathrm{M}(0) = M_0$$

$$icC := \mathrm{c}(0) = \frac{M_0}{\mathrm{V}(0)}$$

Now that the specific set of parameters has been identified and an algorithm for determining the initial value problems has been tested, it is time to combine everything and write the **plotCM** procedure.

```
> plotCM := proc(PARAM  :: { set( name=realcons ) },
>                VOLUME :: identical(V)={realcons,dependent(t)},
>                TRANGE :: identical(t)=range(realcons) )
>  local odeM, odeC, icM, icC, ConsMass, _pM, _pC, NAMES, OPTS;
>  NAMES := map(lhs,{op(PARAM)});
> #  check that all parameters are specified
>  if evalb( NAMES <> {c[i],Q[i],Q[o],M[0]} ) then
>   ERROR(`must specify c[i], Q[i], Q[o] and M[0]`)
>  fi;
> #  default DEplot options
>  OPTS := arrows=NONE, linecolor=BLUE, stepsize=1;
> #  user-specified DEplot options
>  if nargs>3 then OPTS := OPTS, args[4..nargs] fi;
> #  assemble initial value problems from conservation of mass
>  ConsMass := diff( M(t), t ) = Q[i] * c[i] - Q[o] * M(t) / V ;
>  odeM := subs( VOLUME, PARAM, ConsMass );
>  odeC := value( subs( M(t)=c(t)*V, VOLUME, PARAM, ConsMass ) );
>  odeC := op( solve( odeC, { diff( c(t), t ) } ) );
>  icM  := subs( PARAM, M(0) = M[0] );
>  icC  := subs( VOLUME, PARAM, t=0, c(0) = M[0]/V );
> #  create separate plots, then display side by side
>  _pM := DEtools[DEplot]( odeM, M, TRANGE, [[icM]],
>          title=`M (mg) vs. t (min)`, OPTS );
>  _pC := DEtools[DEplot]( odeC, c, TRANGE, [[icC]],
>          title=`c (mg/m^3) vs. t (min)`, OPTS );
>  plots[display]( array( [_pM,_pC] ) );
> end:
```

Text that comes after a **#** is ignored by Maple. These comments, which are not displayed when the procedure is viewed by **print** or **showstat**, are included as aids for anyone who wants to understand **plotCM**, now or in the future. Note that, in addition to processing the three required arguments, this implementation of **plotCM** uses **nargs**, the number of arguments in the current procedure call, and **args**, the list of arguments for the current procedure call, to forward any optional arguments to the **DEplot** commands. The only other special feature of this procedure is the explicit reference to the libraries to

which **DEplot** and **display** belong; this is done to ensure that **plotCM** works even when the complete **DEtools** and **plots** packages have not been loaded in the current session.

4. Refine and implement a solution

The two different reactors can be used to test **plotCM**. For the first reactor, with $c_i = 50$ mg/m^3, $Q_i = Q_o = 5$ m^3/min and an initial accumulation rate of $A = 200$ mg/min, the initial mass of solute is determined from the statement of conservation of mass at $t = 0$:

```
> InitAccum;
```

$$A = Q_i c_i - \frac{Q_o \, \mathrm{M}(0)}{\mathrm{V}(0)}$$

When the initial volume is V_0 = 100 m^3, the initial mass in the reactor must be

```
> solve( subs( A=200, c[i]=50, Q[i]=5, Q[o]=5, V(0)=100, InitAccum ),
        { M(0) } );
```

$$\{\mathrm{M}(0) = 1000\}$$

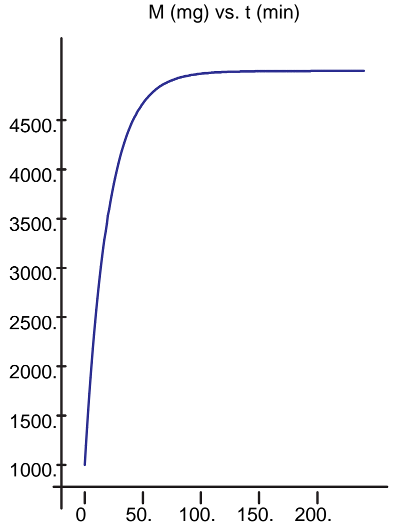
The set of parameter values for the first reactor is

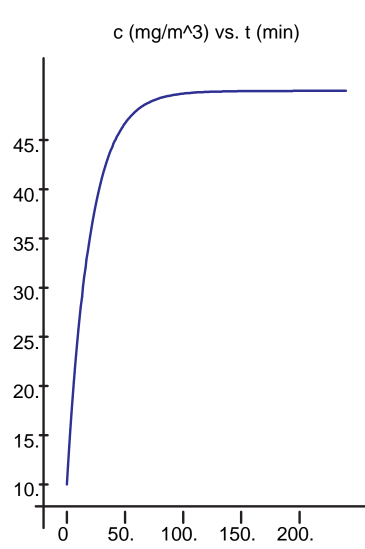
```
> PARAM1 := { c[i]=50, Q[i]=5, Q[o]=5, M[0] = 1000 };
```

$$PARAM1 := \{c_i = 50,\ Q_i = 5,\ Q_o = 5,\ M_0 = 1000\}$$

and, as discussed in Step 2, the volume is always 100 m^3. Thus, the graphs of the mass and concentration during the first 4 hours (240 minutes) of operation can be obtained with the following call to **plotCM**:

```
> plotCM( PARAM1, V = 100, t=0..240 );
```





Note that, except for the scales, these curves appear to have the same qualitative behavior. From their initial values, both the mass and concentration increase rapidly to a constant level. The concentration clearly approaches 50 mg/m^3, which is the value of c_i. Do you understand why this must be the case? The mass approaches 100 times this value, that is, 5 g.

The plots produced by **plotCM** are useful for understanding the behavior of the solution to this one problem. One means of determining if this behavior changes with different initial conditions is to include the direction fields for each differential equation in the plot. Direction fields are produced by **DEplot** when, for example, the **arrows=SMALL** optional argument is specified. Since **plotCM** automatically forwards optional arguments to **DEplot**, it suffices to include this optional argument after the three required arguments for **plotCM**. In the case of this first reactor, you should see that all solutions with initial concentrations below 50 mg/m^3 have the same qualitative behavior. (See Problem 14 at the end of this chapter.)

The second reactor differs only in the output flow rate

```
> PARAM2 := { c[i]=50, Q[i]=5, Q[o]=11/2, M[0] = 1000 };
```

$$PARAM2 := \left\{ Q_o = \frac{11}{2},\ c_i = 50,\ Q_i = 5,\ M_0 = 1000 \right\}$$

Because the input and output flow rates are not equal, the volume is not constant. In this case the volume in the reactor decreases by 1/2 cubic meter per minute. The request for graphs of the mass and concentration during the first 4 hours of operation is made with the single command:

```
> plotCM( PARAM2, V=100-t/2, t=0..240 );
```

```
Error, (in DEtools/DEplot/drawlines) Stopping integration
due to, division by zero
```

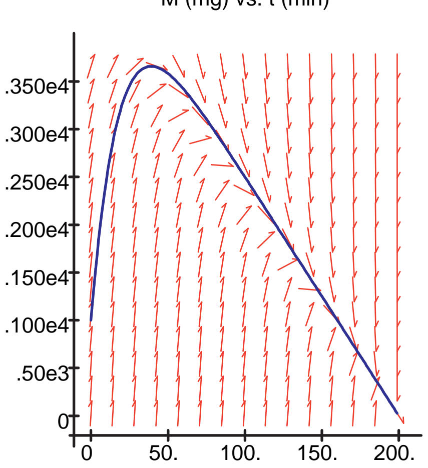
The error message indicates that **DEplot** has encountered division by zero, but does not tell you exactly when or where this occurs. To explain and, if possible, provide a resolution to this problem, recall (from Step 3) that both differential equations include the volume in the denominator of a rational expression:

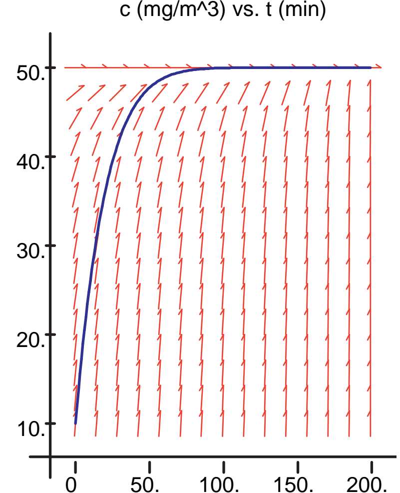
```
> simplify( subs( PARAM2, V(t)=100-t/2, [ odeC, odeM ] ) );
```

$$\left[\frac{\partial}{\partial t}\mathrm{c}(t) = 10\,\frac{\mathrm{c}(t) - 50}{-200 + t},\ \frac{\partial}{\partial t}\mathrm{M}(t) = \frac{-50000 + 250t + 11\mathrm{M}(t)}{-200 + t} \right]$$

Since the volume decreases by 1/2 m^3/min, the tank is completely empty after 200 minutes. Thus, the best that can be hoped for is to plot the solution curves for any time ending before 200 minutes. For example, the solutions (and direction fields) that terminate one minute before the singularity is encountered are:

```
> plotCM( PARAM2, V=100-t/2, t=0..199, arrows=SMALL,
>        dirgrid=[15,15] );
```





The concentration looks essentially the same as in the first reactor. The mass, however, is quite different and will be explored further in the final stage of the five-step process.

5. Verify and test the solution

The analysis concludes by verifying that the graphical solutions are consistent with properties of the differential equations. (Note that although these differential equations are not difficult to solve, the explicit solutions are not used at any stage of the analysis. This is typical of qualitative analysis.)

```
> simplify( subs( PARAM1, V(t) = 100, [ odeM, odeC ] ));
```

$$\left[\frac{\partial}{\partial t}\mathrm{M}(t) = 250 - \frac{1}{20}\mathrm{M}(t),\ \frac{\partial}{\partial t}\mathrm{c}(t) = \frac{5}{2} - \frac{1}{20}\mathrm{c}(t)\right]$$

The fact that the volume is always 100 cubic meters means that $\mathrm{M}(t) = \mathrm{V}(t)\mathrm{c}(t) = 100\,\mathrm{c}(t)$ and so the mass and concentration should have the same qualitative behavior, scaled by a factor of 100.

If the concentration of solute in the reactor is to be constant, then $\mathrm{c}'(t) = 0$ and the concentration must be a zero of the right-hand side of the corresponding differential equation, that is, $\frac{5}{2} - \frac{\mathrm{c}(t)}{20} = 0$. This constant solution, $\mathrm{c}(t) = 50$ mg/m^3, is often called the steady-state solution. The corresponding constant mass solution is $\mathrm{M}(t) = 5000$ mg $= 5$ g. These results are consistent with both the equation and the plots created with **plotCM**. The solutions reach these levels 10 to 20 minutes before the end of the second hour of operation, and remain at these levels as long as the reactor continues to operate.

The analysis is somewhat different for the reactor with different inflow and outflow rates.

```
> simplify( subs( PARAM2, V(t)=100-t/2, [ odeM, odeC ] ));
```

$$\left[\frac{\partial}{\partial t}\mathrm{M}(t)=\frac{-50000+250t+11\mathrm{M}(t)}{-200+t},\ \frac{\partial}{\partial t}\mathrm{c}(t)=10\frac{\mathrm{c}(t)-50}{-200+t}\right]$$

While the steady-state concentration is still $c(t)=50$ mg/m^3, there are no constant solutions to the mass equation. Since the outflow rate exceeds the inflow rate by 1/2 m^3/min, it is not surprising that the reactor eventually becomes dry—this is the source of the singularity after 200 minutes. At this time the reactor ceases to function. This problem is not numerical; it is intimately connected with the specific design of this reactor.

What If

In some situations it is necessary to control the concentration and/or amount of solute in the reactor. For example, if the solute is toxic, Occupational Safety and Health Administration (OSHA) standards may require the manufacturer to keep the concentration under 40 mg/m^3. At the same time, production guidelines may stipulate that the solution is usable only when the concentration exceeds 15 mg/m^3. Propose and critique a plan for adapting the reactor to maintain a constant volume and to satisfy these criteria.

One approach would be to start the reactor with the solute entering at its normal rate and concentration. The first few minutes of outflow would have to be discarded or recycled, and then a usable solution would be produced. Just before the concentration reaches the OSHA safety limit, the source of solute would be interrupted. To maintain a constant volume in the reactor, the inflow would be pure solution. Just before the concentration of the solution reaches the lower (production) threshold, the solute would be added to the inflow solution. This process would then repeat for the remainder of the production run. Try to implement this plan with a sequence of calles to **plotCM** using different parameters and time intervals.

SUMMARY

The final topic in this introduction to Maple has been the Maple programming language, in particular, the definition of Maple procedures using both the arrow operator (**->**) and **proc** ... **end**. The discussion included issues relating to error handling, return values, type-based pattern matching, and local and global variables. The conditional (**if** ... **then** ... **else** ... **fi**) and repetition (**for** ... **while** ... **do** ... **od**) statements were used in a number of examples. Other commands discussed included **showstat**, **select**, **remove**, **map**, **map2**, **interface**, **printlevel**, **debug**, and **maplemint**. The application utilized the principle of conservation of mass to analyze a mixing problem, as a chemical engineer might do.

Key Words

breakpoints
built-in command
concatenation (dot) operator
conditional statement
debugger
environment variables
functional operator
global variables
local variables
Maple debugger
Maple programming language
procedure
random number generator
recursion
remember table
repetition statement
selection, removal
type-based pattern matching
watchpoints

Maple Commands

-> (arrow operator)
. (dot or concatenation operator)
:: (type-based pattern matching)
args, **nargs**
assigned
debug, **trace**, **printlevel**, **maplemint**
ERROR
for ... **from** ... **to** ... **by** ... **while** ... **do** ... **od**
for ... **in** ... **while** ... **do** ... **od**
if ... **then** ... **elif** ... **else** ... **fi**
interface, **verboseproc**
isolate (from **student** package)
local, **global**
map, **map2**
member
NULL (empty expression)
option remember
print
proc ... **end**
rand
RETURN
select, **remove**
showstat

References

1. Chapra, S.C. and Canale, R.P., *Introduction to Computing for Engineers*, 2nd Ed., New York: McGraw-Hill, 1994, pp. 664–672.
2. Lindeburg, M.R., *Engineer-In-Training Reference Manual*, 8th Ed., Belmont, CA: Professional Publications, pp. 10-7 and 10-10.
3. Monagan, M.B., Geddes, K.O., Labahn, G., and Vorkoetter, S., *Maple V Programming Guide*, Springer-Verlag, 1996.
4. Nicolaides, R., and Walkington, N., *Maple: A Comprehensive Introduction*, Cambridge, 1996.

Concluding Remarks

You have now completed this module of the Engineer's Toolkit series. If you have worked through the majority of the Examples, Try It!'s, Applications, and Problems in each chapter, you should now have a good understanding of Maple and some of its uses in engineering. More importantly, you should have the knowledge and confidence to know you can extend your knowledge to tackle problems that might arise in areas and applications other than those presented here.

Congratulations, and good luck!

Exercises

1. (a) Write a Maple procedure, **TIMING**, that reports the time needed to evaluate the argument, then returns a list whose first element is the elapsed time and all subsequent elements are the results, if any, of evaluating the argument. The input argument will need to be specified in single quotes (**'**) to prevent evaluation of the argument before the appropriate time in the body of the procedure. (Note that there are several differences between **TIMING** and the built-in **time** command; in fact, **TIMING** will use **time**.)

   (b) Use **TIMING** to compare three implementations of a procedure that selects all input arguments that are both a perfect square and a perfect cube. Rank the implementations in terms of efficiency. Does this ranking depend on the size of the numbers or number of elements in the set?

```
> ispower := (NUMBER::integer,POWER::posint) ->
              type( surd(NUMBER,POWER), integer ):
> squares := (SET :: set(integer)) -> select( ispower, SET, 2 ):
> cubes  := (SET :: set(integer)) -> select( ispower, SET, 3 ):
> SCselect1 := ( SET::set(integer)) -> squares( SET ) intersect
                                        cubes( SET ):
> SCselect2 := (SET::set(integer)) -> squares( cubes( SET ) ):
> SCselect3 := (SET::set(integer)) -> cubes( squares( SET ) ):
```

2. Write a Maple procedure, **plotD**, that plots a function (of one variable) and each of its first two derivatives on separate graphs. In order to make the three plots appear side-by-side, display a 1×3 array of plots. Include appropriate titles for each plot.

   The following three commands should all produce the expected results

```
> plotD( arcsin(x), x=-1..1 );
> plotD( 1/(1+x^2), x=-5..5, color=BLUE, style=POINT );
> plotD( {sin(x),cos(x)}, x=0..2*Pi );
```

   Note that only two arguments are required; additional arguments should be passed directly to the plot commands (see **?args** for more information about argument lists).

3. Consider the function **Fib** implemented in Example 7-11. The **else** part of the conditional is executed the first time the function is called with an argument of 1 or 0. The same effect can be achieved by explicitly entering these results into the remember table (via standard assignments). This suggests a third implementation for the Fibonacci numbers:

```
> FIB := proc( n::nonnegint )
>   option remember;
>   RETURN( FIB(n-1) + FIB(n-2) )
> end:
> FIB(0):=0:
> FIB(1):=1:
```

   (a) Verify that **FIB** correctly computes the Fibonacci numbers.
   (b) Which of **fib**, **Fib**, and **FIB** is most efficient? (Use **TIMING** to compute f_{20}.) See the **forget** help worksheet for information on resetting remember tables.
   (c) What happens if the two assignments are moved before the definition of **FIB**?
4. Write an alternate version of **minpos** (origanally defined in Example 7-14) that does not utilize the repetition command. Use **TIMING** to compare the efficiency of the two implementations.
5. A shaft-position encoder provides a 3-bit signal indicating the position of a motor shaft in steps of 45 degrees using the following code:

Shaft Position	Encoder Output
0–45	000
46–90	001
91–135	010
136–180	011
181–225	100
226–270	101
271–315	110
316–360	111

   (a) Write a Maple procedure that produces the appropriate encoder output for a shaft position, or list of shaft positions.
   (b) Write separate Maple procedures that correspond to binary encoders that are ON, that is, returns the value 1, when i) the shaft is in the second half of its rotation cycle, ii) the motor shaft is not in the first eighth of its cycle and iii) either in the first or third quarter of its cycle.

   Each encoder should contain all necessary type and error checking. The **rand** command can be used to generate a random number generator of integers between 0 and 360. For example, one way to generate data for this problem would be as follows:

```
> signal := rand(0..360):
> pos := [ seq( signal(), i=1..10 ) ];
```

$$pos := [359, 165, 327, 194, 316, 251, 226, 90, 312, 66]$$

6. The Chebyshev polynomials of the first kind, $T_n(x)$, satisfy the recurrence relations $T_n(x) = 2x T_{n-1}(x) - T_{n-2}(x)$ for $n \geq 2$ with $T_0(x) = 1$ and $T_1(x) = x$.
   (a) Write a recursive definition of the Chebyshev polynomials.
   (b) Find Maple's built-in Chebyshev functions. How are these functions implemented?
7. Implement a Maple procedure, **bodeampplot**, that creates the corrected Bode amplitude plot for a frequency response function. Your procedure should reproduce the corrected Bode amplitude plot for the circuit in the Electric Filter application (Chapter 6) with $L = 10$ H, C = 6 μF, and R = 1 kΩ with the following commands:

```
> PARAM := [ L = 10, C = 6*10^(-6), R = 10^3 ];
> RESPONSE := j*R/(L^2*C*omega^3 - j*R*C*L*omega^2 - 2*L*omega + j*R);
> omega0 := subs( PARAM, 1/sqrt(L*C) );
> bodeampplot( RESPONSE, omega=omega0/10..omega0*10, PARAM,
>              axes=FRAMED, labels=['z','gain'],
>              title='Corrected Bode Amplitude Plot' );
```

8. (a) Fuzzy logic was introduced in problem 8 (Chapter 3). Write a Maple function, **mu**, so that **mu**$(A) = \mu_A$, where $\mu_A(x) = \dfrac{1}{1+(x-A)^2}$ is the membership function for the element A.
   Hint: Use the arrow operator.
   (b) Write Maple implementations of the three fuzzy logical operations **fuzzyNOT**, **fuzzyOR**, **fuzzyAND**.
   Hint: Use **convert,piecewise** to simplify the maximum and minimum of a collection of functions.
   (c) Use the procedures created in (a) and (b) to find the membership functions for $A = \{10\}$, $B = \{15\}$, $C = \overline{A}$, $D = A \cup B$, $E = A \cap B$, $F = \{ 5, 10, 15, 20 \}$, and $G = \overline{\{ 5, 15, 20 \}}$. Plot each of these membership functions.
9. This is a continuation of the Try It! at the end of Section 3.2 and problem 9 of Chapter 3.
   Write an analog-to-digital converter, **d2a**, that accepts an analog signal from a specified range and returns the appropriate digital code as a binary number. The three input arguments to the procedure should be: the analog signal (either a single number or a list of numbers), the range of valid signals, and the number of digital codes to be used to represent the code. All output codes should have the same length.
   Be sure that your procedure includes all appropriate error checking. Select, and implement, a reasonable way of handling data that is outside the range of expected signals. (It is not acceptable for the procedure to simply report an error and quit.)

Test this procedure using the following sets of commands:

```
> analog := rand( 0 .. 5000 ) / 1000:  # random numbers between 0 and 50
> signal := map( evalf, [ seq(analog(), k=1..100) ] ):
> d2a( signal, 0..50, 128 );
>
> noisy := rand( -100 .. 600 ) / 10:  # random numbers between -10 and 60
> badsignal := map( evalf, [ seq(noisy(), k=1..10 ) ] ):
> d2a( noisy, 0..50, 128 );
> d2a( noisy, -10..60, 128 );
```

10. Modify the **plotCM** procedure to accept an optional argument specifying the specific plots to be included in the output. For example, specifying **plotlist = [M]** would produce only the mass plot and **plotlist = [C]** only the concentration plot; the default would be equivalent to **plotlist = [C,M]**. Be sure to include appropriate error checking.
11. As part of the process in manufacturing an eye wash product, a boric acid solute enters a tank filled with purified water at time $t = 0$ minutes. In practice it is not possible to achieve a constant inflow concentration. One way to model this is to assume the input concentration varies sinusoidally with time. That is, instead of c_i being a constant, use

$$c_i(t) = \bar{c}_i + \bar{c}_a \sin\left(\frac{2\pi t}{T}\right).$$

where the average inflow rate is $\bar{c}_i = 50$ mg/m^3, the amplitude of the inflow variation is $\bar{c}_a = 40$ mg/m^3, and $T = 50$ min is the period. (Assume all other parameters are the same as in the reactor with constant volume.)

(a) Plot the concentration and total mass of boric acid produced by this tank as a function of time.

(b) Does this reactor approach the original (constant inflow) reactor as $\bar{c}_a \to 0$?

(c) How much boric acid is in the tank after three hours? How much boric acid has been produced during this time period?

12. The analysis of the reactor model conducted in the application focused solely on the concentration levels in the reactor. Another aspect of this problem is the quality and quantity of the solution obtained from the reactor.

(a) Show that the total amount of solution produced during the first t minutes, $\mathrm{p}(t) = \int_0^t Q_o\, \mathrm{c}(\tau)\, d\tau$, satisfies each of the following differential equations:

i) $\mathrm{p}'(t) = Q_0\, \mathrm{c}(t)$

ii) $\mathrm{p}''(t) + \dfrac{Q_i}{V_0}\mathrm{p}'(t) = \dfrac{Q_o Q_i c_i}{V_0}$

(Assuming the volume in the reactor is held constant.)

(b) Find appropriate initial conditions for each of the equations in (a).

(c) Once the concentration has essentially reached its steady-state solution the production curve is essentially linear. What is the slope of this line?

13. A tank with two input pipes and one output pipe contains 100 gallons of pure water at time $t = 0$. Pure water enters the tank through one of the input pipes at a flow rate of 0.5 gallons per minute. Sugared water having concentration $c_i = 0.5$ lbm/gal enters through the other input pipe at a flow rate of 1 gallon per minute. The perfectly mixed solution drains from the tank at a flow rate such that the level of liquid in the reactor is constant. Find the time when the concentration in the drain mixture will be exactly 0.3 lbm of sugar per gallon. Without doing any calculations, what is the steady-state concentration of the drain mixture? At what approximate time will this occur? (This problem is based on a problem in *Engineer-In-Training Reference Manual*, 8th Ed.)

NOTE: 1 gallon of liquid is equivalent to 0.0037854 m^3 and 1 lbm is equivalent to 0.4536 kg.

(a) Use a direction field to verify the claim in step 4 of the solvent and solute application that all solutions with initial concentration below 50 mg/m^3 exhibit the same qualitative behavior.

(b) Determine the qualitative behavior of the mass and concentration of solute in the first reactor considered in the application when the initial concentration exceeds 50 mg/m^3.

(c) What is special about the initial concentration of 50 mg/m^3?