Chapter 4

Exponential Growth

§4.1 Powers

User-defined function, `function` declarations, preconditions and post conditions, parameter lists, formal and actual parameters, functions that call other functions, scope rules, development through generalization.

§4.2 Binomial Coefficients

Weakening the precondition

There are a number of reasons why the built-in `sin` function is so handy. To begin with, it enables us to compute sines *without having a clue* about the method used. It so happens that the design of an accurate and efficient sine function is somewhat involved. But by taking the "black box" approach, we are able to be effective `sin`-users while being blissfully unaware of how the built-in function works. All we need to know is that `sin` expects a real input value and that it returns the sine of that value interpreted in radians.

Another advantage of `sin` can be measured in keystrokes and program readability. Instead of disrupting the "real business" of a program with lengthy compute-the-sine fragments, we merely invoke `sin` as required. The resulting program is shorter and reads more like traditional mathematics.

Most programming languages come equipped with a *library* of built-in functions. The designers of the language determine the library's content by anticipating who will be using the language. If that group includes scientists and engineers, then invariably there will be built-in functions for the sine, cosine, log, and exponential functions because they are of central importance to work in these areas.

It turns out that if you need a function that is not part of the built-in function library, then *you can write your own.* The art of being able to write efficient, carefully organized functions is an absolutely essential skill for the computational scientist because it suppresses detail and permits a higher level of algorithmic thought.

To illustrate the mechanics of function writing we have chosen a set of examples that highlight a number of important issues. On the continuous side we look at powers, exponentials, and logs. These functions are monotone increasing and can be used to capture different rates of growth.

Factorials and binomial coefficients are important for counting combinations. We bridge the continuous/discrete dichotomy through a selection of problems that involve approximation.

4.1 Powers

To raise x to the n-th power, in MATLAB we use the expression `x^n`. Some programming languages, however, do not include a power operator and a programmer would have to write a code fragment, such as the one below, to evaluate x^n:

```
xpower= 1;
for k= 1:n
   xpower= x*xpower; %{xpower = x^k}
end
```

Each pass through the loop raises the "current" power of x by one. Without the power operator, it is not unreasonable to insert this single-loop calculation as required in a program that requires the computation of a power in just a few places. However, it is not hard to imagine a situation where exponentiations are required many times throughout a program. It is then a major inconvenience to be personally involved with each and every powering if the power operator is not available. The script `Example4_1` reinforces the point. The program illustrates the kind of tedium that is involved when the same computation is repeated over and over again. There is a threefold application of the exponentiation "idea." If we didn't know about MATLAB's built-in power operator, then we would like to specify once and for all how powers are computed and then just use that specification to get x^n, y^n, and z^n without repeating any code.

Fortunately, there is a way to do this and it involves the creation and use of a *programmer-defined function*. The concept is illustrated in `Example4_2`. In this program, the function has the name `pow` and, after its creation, it is *invoked*, or referenced, by script file `eg4_2.m`. The function is saved in a file separate from the script file and it has a filename that is the same as the function name. The function filename also has the extention `.m`.

Let us look at how a function is structured. A casual glance at the function code

```
function apower = pow(a, n)
% Post:  apower=a^n
% Pre:  n>=0

apower= 1;
for k= 1:n
   apower= apower*a;   % {apower=a^k}
end
```

shows that a function resembles other scripts that we have written. The last few lines are familiar-looking code that comprises the function body. However, the function begins with a line of code that contains the keyword `function`, the function name, an *output argument list*, and an *input parameter list*. This line of code is called the *function header*:

$$\textbf{function}\quad \underbrace{\texttt{apower}}_{\textit{output argument}} \quad = \quad \underbrace{\texttt{pow}}_{\textit{function name}} \quad (\quad \underbrace{\texttt{a , n}}_{\textit{input parameter list}} \quad)$$

```
% Example4_1:  {Examines |x^n + y^n - z^n| for real x,y,z and whole number n.}

x= input('Enter x:  ');
y= input('Enter y:  ');
z= input('Enter z:  ');
n= input('Enter nonnegative integer n:  ');

xpower= 1;
ypower= 1;
zpower= 1;
for k= 1:n
   % {xpower=x^k; ypower=y^k; zpower=z^k}
   xpower= xpower*x;
   ypower= ypower*y;
   zpower= zpower*z;
end

value= abs(xpower+ypower-zpower);
fprintf('|x^n + y^n - z^n| = %f\n', value)
```

Output:

```
Enter x:  3
Enter y:  2
Enter z:  5
Enter nonnegative integer n:  3
|x^n + y^n - z^n| = 90.000000
```

The input parameter list is made up of the function's *formal parameters.* The function `pow` has two formal parameters: `a` and `n`. A parameter is like a variable in that it is a named memory space that stores a value. The parameter list is enclosed in parentheses and separated by commas, or if there are no parameters then the parentheses will be empty. Formal parameters are sometimes called *arguments.* Thus, `pow` is a 2-argument function. Our function `pow` returns one value through the output argument name `apower`. If a function returns multiple values, we put the output arguments in a comma-separated list enclosed by square brackets [ ].

After the function header comes the *specification.* This is a comment that communicates all one needs to know about using the function. It has a *post-condition* part and a *pre-condition* part identified with the abbreviations "Post" and "Pre":

```
function apower = pow(a, n)
% Post:  apower=a^n
% Pre:  n>=0
```

The post-condition describes the value that is produced by the function. To say that the post-condition is "`a^n`" is to say that the function returns the value a^n. In other words, the post-condition describes the results or the effects of the function.

Function file `pow.m`:

```
function apower = pow(a, n)
% Post:  apower=a^n
% Pre:  n>=0

apower= 1;
for k= 1:n
   apower= apower*a;   % {apower=a^k}
end
```

Script file `eg4_2.m`:

```
% Example4_2:  {Examines |x^n + y^n - z^n| for real x,y,z and whole number n.}

x= input('Enter x:  ');
y= input('Enter y:  ');
z= input('Enter z:  ');
n= input('Enter nonnegative integer n:  ');

value= abs(pow(x,n) +pow(y,n) - pow(z,n));
fprintf('|x^n + y^n - z^n| = %f\n', value)
```

Output:

```
Enter x:  3
Enter y:  2
Enter z:  5
Enter nonnegative integer n:  3
|x^n + y^n - z^n| = 90.000000
```

The pre-condition indicates properties that must be satisfied for the function to work correctly. Apparently, `pow` does not work with negative n. Since there is no restriction on the value of `a`, there is no mention of this parameter in the precondition.

The specification should *not* detail the method used to compute the returned value. The goal simply is to provide enough information so that a user knows how to use the function. MATLAB uses the convention where the first comment line below the function header contains a succinct, one-line description of the function. A complicated function requiring a lengthy description should have a simplified one-line comment that summaries its purpose followed by the more detailed post- and pre-conditions[1].

Now let's go "inside" the function and see how the required computation is carried out. To produce a^n, function `pow` needs its own variables, `apower` and `k`, to carry out the looping and repeated multiplication. In this regard, function `pow` is just like the main script `eg4_2` which has *its* own variables. To stress the distinction between the main script's variables and those used inside a function, we refer to the latter as *local variables*. Thus, `apower` and `k`, and indeed the

---

[1] MATLAB's `lookfor` and `help` commands work with user-defined as well as built-in functions. The command `lookfor` ⟨*subject word*⟩ will display the names of all the functions in the search path that contains the subject word in the first comment line in a function. The command `help` ⟨*function name*⟩ displays the lines of comments that immediately follow the function header up to the first blank line.

parameters `a` and `n`, are *local* to `pow`.

Finally, we have reached the *body* of `pow` where the recipe for exponentiation is set forth:

```
apower= 1;
for k= 1:n
   apower= apower*a;   % {apower=a^k}
end
```

The function body is the part of the function that follows the function header and the specification comments. Notice how the powers are built up in `apower` until the required a^n is computed. Since `apower` is specified to be the *output argument* in the function header, the value of `apower` after execution of the function body is *returned* to the place in the script `eg4_2` that invoked function `pow`.

The last item on our agenda concerns the use of `pow` by the main program. Recall that a built-in function such as `sin` returns a value that can be used in an arithmetic expression, e.g., `v= sin(3*x) + 4*sin(2*x)`. The same is possible with `pow`:

$$\mathtt{d} = \underbrace{\mathtt{pow(x,n)}}_{x^n} + \underbrace{\mathtt{pow(y,n)}}_{y^n} + r\,\underbrace{\mathtt{pow(z,n)}}_{z^n}$$

In any arithmetic expression that calls for a power, we merely insert an appropriate reference to `pow`. These references are called *function calls.* The assignment to `d` includes three function calls to `pow`. Suppose `x`, `y`, `z`, and `n` have value 2, 5, 4 and 3 respectively. It follows that

$$\left.\begin{array}{l}\mathtt{pow(x,n)}\\ \mathtt{pow(y,n)}\\ \mathtt{pow(z,n)}\end{array}\right\}\text{ has the value }\left\{\begin{array}{l}2^3 = 8\\ 5^3 = 125\\ 4^3 = 64\end{array}\right.$$

and so `d` is assigned the value 8 + 125 + 64 = 197.

Every time a function is referenced, make sure that the number of arguments and their type agree with what is specified in the function header. Choose function names that are descriptive and unique. We choose the function name `pow` in order to distinguish it from a MATLAB built-in function called `power`[2]. You can access a function that you have defined if it is in the *current working directory* or if the directory in which the function is stored is on the *search path*[3]. To put a directory onto the search path, use MATLAB's menu option *File → Set Path.*

A function like `pow` is conveniently thought of as a factory. The "raw materials" are a and n and the "finished product" is a^n. Thus, an "order" to produce $(2.5)^3$ involves (a) the receipt of the 2.5 and 3, (b) the production of the "consumer product" 2.5^3, and (c) the "shipment" of the computed result 15.625. See FIGURE 4.1.

An important substitution mechanism attends each function call and it is essential that you master the underlying dynamics. We motivate the discussion by considering how we use the Centigrade-to-Fahrenheit formula

$$F = \frac{9}{5}C + 32.$$

---

[2] How do you know if a "word" is a MATLAB function name? The command `help` ⟨*word*⟩ will list any documentation associated with ⟨*word*⟩.

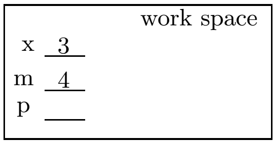
[3] Since there is only one current working directory at one time, you can create functions of the same name in different directories. If you "must" create a function that has the same name as a built-in function, read MATLAB's *Help* documentation under the topics *function* and *search path* for implementation details.

If we substitute "20" for "C" and evaluate the result, then we conclude that $F = 68$. The formula is merely a template with F and C having placeholder missions.

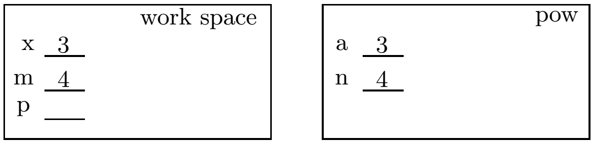
The situation is similar in a function. The function is merely a formula *in algorithmic form* into which values are substituted. Let's step through a call to `pow` and trace what happens. Consider the following main program fragment:

```
x= 3;
m= 4;
p= pow(x,m);
```

When you run a MATLAB program, the variables in the script are stored in the *work space.* Thus after the first two assignment statements we have the following situation:



Next comes the reference to `pow`. To trace what happens, we draw a "function box" for the memory space used by the function. At the time of the function call, values are passed to function `pow` so parameters (variables) are created *in the function* to hold the passed values:



Notice that the function box is *separate* from the work space used by the script—function parameters and variables are *local* to the function and are unknown to the script's work space. Now execution continues *inside the function box.* At the end of the first pass through the loop in `pow` we reach the following state:

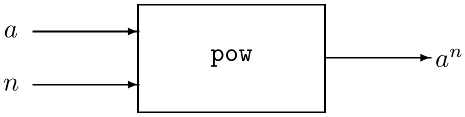
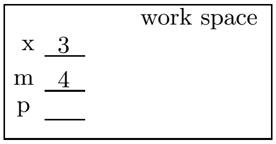
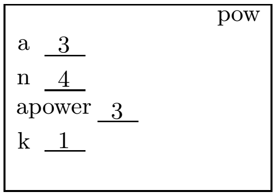
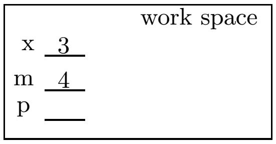


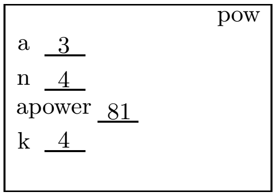
FIGURE 4.1 *Visualizing a Function*



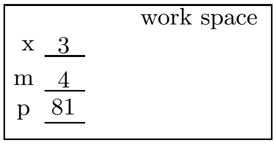


Execution inside `pow` continues until at the end of the fourth and final pass we obtain:





At this time, the function returns the value of the output argument, 81, to the place that has called the function. Now the function box "closes" and control is passed back to the script with the value 81 placed in `p`:



Note that once the function box closes, its variables are lost. A subsequent call to function `pow` will start *without* any memory of the previous function call[4].

PROBLEM 4.1. Note that x^{64} can be obtained through repeated squaring:

$$x \to x^2 \to x^4 \to x^8 \to x^{16} \to x^{32} \to x^{64} .$$

Thus, x^{64} can be "reached" with only six multiplications in contrast to the sixty-three products that are required if we go down the repeated multiplication path:

$$x \to x^2 \to x^3 \cdots \to x^{63} \to x^{64} .$$

Using the repeated squaring idea, write a function `twoPower(k,x)` that computes x^n where $n = 2^k$. Write a good specification.

---

[4] Special declarations may be used to change the properties of a variable. MATLAB allows a variable's value to be preserved between function calls by using the variable declaration `persistent`. A variable may be shared by multiple scripts and functions by using the variable declaration `global`.

PROBLEM 4.2. The Fibonacci numbers $f_1, f_2, \ldots$ are defined as follows:

$$f_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ f_{n-1} + f_{n-2} & \text{if } n \geq 3 \end{cases} . \qquad (1)$$

It can be shown that

$$f_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) . \qquad (2)$$

Write a program that prints a table showing the first 32 Fibonacci numbers computed in two ways. In the first column of the table should be the values produced by the recursion (1) and in the second column the values obtained by using (2). In the latter case, make effective use of `power` and print the real values to six decimal places so that the effects of floating point arithmetic can be observed.

PROBLEM 4.3. A general exponentiation function can be obtained by exploiting the formula

$$x^y = (e^{\ln(x)})^y = e^{\ln(x)*y},$$

implemented in function `powerGR` below. Using `powerGR`, write a program that discovers the smallest positive integer n so that $(\sqrt{n})^{\sqrt{n+1}}$ is larger than $(\sqrt{n+1})^{\sqrt{n}}$.

```
function p = powerGR(x, y)
% Post:  p= x^y
% Pre:  x>0
p= exp(ln(x)*y);
```

PROBLEM 4.4. How big must `n` be before the following fragment prints `inf`?

```
x= 1;
for k= 1:n
   x= powerGR(2,x);
end
disp(x)
```

Try to guess the answer before writing a program to confirm it.

It is possible for one programmer-defined function to call another programmer-defined function. To illustrate, `Example4_3` prints a table of values for the function

$$f(x) = x^n(1-x)^m$$

for the case $n = 3$ and $m = 4$. Function `pow` was shown in Example 4_2 and is excluded from the Example 4_3 box.

A problem-solving strategy known as *function generalization* often leads to a situation where one function calls another. Consider the a^n problem again where now n can be *any* integer. Mathematically, there is no problem if $a \neq 0$ whenever $n < 0$. Suppose for some reason we find the nonnegative n case "easy" and the negative n case "hard." We proceed to develop `pow` as above with the precondition $n \geq 0$. We the realize through the formula

$$a^{-n} = \frac{1}{a^n}$$

Script file `eg4_3.m`:

```
% Example4_3:  The function x^n (1-x)^m

fprintf('Let f(z) = z^(1-z)^4 \n\n');
fprintf(' z f(z) \n');
fprintf('------------------\n');
for z= 0:0.1:10
    fz= product(z,3,4);
    fprintf('%.2f \t %f\n', z, fz);
end
```

Function file `product.m`:

```
function p = product(z, n, m)
% Post:  (z^n)*(1-z)^m
% Pre:  0<=z<=1, m>=0, n>=0
if  n <= m
    p= pow(z*(1-z),n)*pow(1-z,m-n);
else
    p= pow(z*(1-z),m)*pow(z,n-m);
end
```

Output:

```
Let f(z) = z^3 (1-z)^4

    z          f(z)
  -------------------
  0.000    0.00000000
  0.100    0.00065610
  0.200    0.00327680
  0.300    0.00648270
               ⋮
```

that a-to-a-negative-power is merely the reciprocal of a raised to the corresponding positive power:

$$3^{-4} = \frac{1}{81}.$$

This suggests that we build our more general power function upon the restricted version already developed:

```
function apower = powerG(a, n)
% Post:  apower=a^n
% Pre:  a is nonzero if n is negative

if  n>=0
   apower= pow(a,n);
else
   apower= 1/pow(a,-n);
end
```

Then any program that uses `powerG` also needs to access `pow`. Later, after the negative n case is well enough understood, we may dispense with the reference to `pow` and handle the repeated multiplication explicitly. For example, we can compute $a^{|n|}$ and then reciprocate the result based upon a check of n's sign:

```
function apower = powerG(a, n)
% Post:  apower=a^n
% Pre:  a is nonzero if n is negative

apower= 1;
for k= 1:abs(n)
   apower= apower*a;   % {apower=a^k}
end
if  n<0
   apower= 1/apower;
end
```

With this implementation, a program that references `powerG` does not need access to `pow`

PROBLEM 4.5. Make the following three modifications to `Example4_3`. (a) Add a function

```
function dp = derProd(z, n, m)
% Post:  Derivative of x^n (1-x)^m at x=z
% Pre:  m,n>0
```

(b) Modify the program so that it prints a table reporting the values $f(z)$ and $f'(z)$ for $z = 0.0, 0.1, \ldots, 0.9, 1.0$ where $f(x) = x^2(1-x)^3$.

4.2 Binomial Coefficients

Combinatorics is a branch of discrete mathematics that is concerned with the counting of combinations. The simplest combinatoric function is the factorial function $n! = 1 \cdot 2 \cdots (n-1) \cdot n$. The number of ways that n people can stand in a line is given by $n!$. If $n = 4$ and a, b, c, and d designate the four individuals, then here are the $24 = 1 \cdot 2 \cdot 3 \cdot 4$ possibilities:

abcd abdc acbd acdb adbc adcb
bacd badc bcad bcda bdac bdca
cabd cadb cbad cbda cdab cdba
dabc dacb dbac dbca dcab dcba

The factorial function grows very rapidly:

n	$n!$
8	40320
9	362880
10	3628800
$\vdots$	$\vdots$
19	121645100408832000
20	2432902008176640000
21	51090942171709440000

If we write a function for computing $n!$, then we must consider what is the largest n that can be used given MATLAB's use of double precision number. It turns out the $n = 21$ is the largest acceptable input value and we obtain

```
function f = fact(n)
% Post:  f=n!
% Pre:  0<=n<=12

f= 1;
for k= 1:n
   f= f*k; %{f=k!}
end
```

Reasonable approximations of "large-n" factorials can be obtained via the *Stirling formula*:

$$n! \approx S_n \equiv \sqrt{2\pi n}\left(\frac{n}{e}\right)^n .$$

If we encapsulate this estimate in the form of a function and use our previously created function `pow`, then we obtain

```
function s = stirling(n)
% Post:  s=Stirling approximation to n!
% Pre:  n>=0

if  n==0
```

```
      s= 1;
   else
      s= sqrt(2*pi*n)*pow((n/exp(1)),n);
   end
```

The case $n = 0$ is handled separately and is included to simplify the use of `stirling`.

The program `Example4_4` compares the Stirling approximation with the exact value of the factorial function for $n = 0, 1, \ldots, 21$. For this range of n, the relative error in the Stirling approximation S_n is about one percent.

`Example4_4` makes use of three programmer-defined functions: `fact`, `stirling`, and `pow` (through the call to `stirling`). This can be done as long as all the programmer-defined functions are in the *current working directory* or in a directory that is on the *search path.*

PROBLEM 4.6. If a positive integer x is written in base-10 notation, then the number of digits required is given by 1 plus the trunc of $\log_{10} x$. Using the identity

$$\log_{10}(n!) = \sum_{k=1}^{n} \log_{10}(k),$$

but without MATLAB's built-in functions, complete the following function:

```
function digits = factorialDigits(n)
% Post:  The number of digits in n!
% Pre:  n>=1
```

Write a program that uses `factorialDigits` and prints a 50-line table. On line n, the table should contain the following values: n, the number of digits in $n!$ as determined by `FactorialDigits`, the number of digits in the Stirling approximation S_n, and S_n. Make use of the user-defined functions in this Chapter.

PROBLEM 4.7. Complete the following function

```
function nni = f(n,d)
% Post:  nni = The number of nonnegative integers <=n that end with the digit d
% Pre:  n>=1, 0<=d<=9
```

Using function `f`, write a program that prints a 20-line table. On line k should appear k and the smallest n so that $n!$ is divisible by 10^k. $k = 1, 2, \ldots, 20$. Hint: Think about $f(n, 0) + f(n, 5)$.

The number of ways that k objects can be selected from a a set of n objects is given by the binomial coefficient

$$\left(\begin{array}{c} n \\ k \end{array} \right) \equiv \frac{n!}{k!(n-k)!}.$$

Thus, there are

$$\left(\begin{array}{c} 5 \\ 2 \end{array} \right) = \frac{5!}{2!3!} = \frac{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5}{(1 \cdot 2)(1 \cdot 2 \cdot 3)} = \frac{5 \cdot 4}{1 \cdot 2} = 10$$

```
% Example4_4:  Stirling Approximation

fprintf(' n\t n!\t Stirling Appx.\n')
fprintf('----------------------------------------------------\n')
for n= 0:21
   fprintf('%2d\t%21.0f\t%21.0f\n', n, fact(n), stirling(n))
end
```

Output:

```
 n                    n!         Stirling Appx.
-----------------------------------------------
 0                     1                      1
 1                     1                      1
 2                     2                      2
 3                     6                      6
 4                    24                     24
 5                   120                    118
 6                   720                    710
 7                  5040                   4980
 8                 40320                  39902
 9                362880                 359537
10               3628800                3598696
11              39916800               39615625
12             479001600              475687486
13            6227020800             6187239475
14           87178291200            86661001741
15         1307674368000          1300430722199
16        20922789888000         20814114415223
17       355687428096000        353948328666100
18      6402373705728000       6372804626194297
19    121645100408832000     121112786592293600
20   2432902008176640000    2422786846761128960
21  51090942171709440000   50888617325509492736
```

possible chess matches within a pool of 5 players. Similar cancelations permit us to compute

$$\left(\begin{array}{c} 26 \\ 4 \end{array}\right) = \frac{26!}{4!22!} = \frac{26 \cdot 25 \cdot 24 \cdot 23}{1 \cdot 2 \cdot 3 \cdot 4} = 14950$$

which is the number of four-letter words with distinct letters and

$$\left(\begin{array}{c} 52 \\ 5 \end{array}\right) = \frac{52!}{5!47!} = \frac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5} = 2,303,000$$

which is the number of 5-card poker hands. From the definition we may write

```
function bc = binCoeff0(n,k)
% Post:  Number of ways to select k objects from a set of n objects
% Pre:  0<=k<=n<=21
bc= fact(n)/fact(k)/fact(n-k);   %{b = n choose j}
```

The trouble with this implementation is that n cannot exceed 21 because the function `fact` breaks down for larger values. However, from the above examples we see that there is considerable cancelation between the factorials in the numerator and denominator. Indeed, it can be shown that

$$\left(\begin{array}{c} n \\ k \end{array}\right) \equiv \frac{n \cdot (n-1) \cdots (n-k+1)}{1 \cdot 2 \cdots k}.$$

Using this formula for the binomial coefficient we obtain

```
function bc = binCoeff(n,k)
% Post:  bc = Number of ways to select k objects from a set of n objects
% Pre:  0<=k<=n<=53

bc= 1;
for j= 1:k
   bc= bc * (n-j+1) / j;
end
```

The restriction that n be less than or equal to 53 has to do with the number of accurate digits in a double precision number. The program `Example4_6` prints out an array of binomial coefficients.

Binomial coefficients arise in many situations. The term itself comes from the fact that if

$$(x+y)^n = a_0 x^n + a_1 x_{n-1} + a_2 x_{n-2} y^2 + \cdots + a_{n-1} x y^{n-1} + a_n y^n,$$

then

$$a_k = \left(\begin{array}{c} n \\ k \end{array}\right) \qquad k = 0, \ldots, n.$$

For example,

$$\begin{aligned} (x+y)^4 &= x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4 \\ &= \left(\begin{array}{c} 4 \\ 0 \end{array}\right) x^4 + \left(\begin{array}{c} 4 \\ 1 \end{array}\right) x^3 y + \left(\begin{array}{c} 4 \\ 2 \end{array}\right) x^2 y^2 + \left(\begin{array}{c} 4 \\ 3 \end{array}\right) x y^3 + \left(\begin{array}{c} 4 \\ 4 \end{array}\right) y^4 \end{aligned}$$

```
% Example4_6:  The Pascal Triangle of binomial coefficients

nmax= 10; % highest n value
for n= 0:nmax
   for k= 0:n
      fprintf('%5d ', binCoeff(n,k))
   end
   fprintf('\n')
end
```

Output:

```
    1
    1    1
    1    2    1
    1    3    3    1
    1    4    6    4    1
    1    5   10   10    5    1
    1    6   15   20   15    6    1
    1    7   21   35   35   21    7    1
    1    8   28   56   70   56   28    8    1
    1    9   36   84  126  126   84   36    9    1
    1   10   45  120  210  252  210  120   45   10    1
```

PROBLEM 4.8. For a given n, the binomial coefficient $\binom{n}{k}$ attains its largest value when `k = ceil(n/2)`. Write a program that prints the value of `binCoeff(n,ceil(n/2))` for n = 1 to 30. Explain why an incorrect value is returned when $n = 30$.

PROBLEM 4.9. Let S_n be the Stirling approximation to $n!$ and define $\beta(n, k)$ to be the approximation of "n choose k":

$$\beta(n,k) = \frac{S_n}{S_k S_{n-k}}$$

Assume that $S_0 = 1$. Write a function `stirlingBC(n)` that returns $\beta(n, k)$. Make effective use of `stirling(n)`. Write a program that uses `stirlingBC` to compute

$$e_n = \max_{0 \le k \le n} \frac{\left|\beta(n,k) - \binom{n}{k}\right|}{\binom{n}{k}}$$

for $n = 1$ to 30. Make use of the user-defined functions in this Chapter.

PROBLEM 4.10. Note that

$$\binom{n}{k} = \binom{n}{n-k}.$$

Modify `BinCoeff` so that it uses the expression on the right hand side if $2k > n$. Rerun `Example4_6` with the modified function. Explain why the modified program is more efficient. Make use of the user-defined functions in this Chapter.

PROBLEM 4.11. Imagine writing n letters and addressing (separately) the n envelopes. The number of ways that all n letters can be placed in incorrect envelopes is given by the Bernoulli-Euler number

$$B_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} (n-k)!$$

Thus,

$$B_4 = \binom{4}{0} 4! - \binom{4}{1} 3! + \binom{4}{2} 2! - \binom{4}{3} 1! \binom{4}{4} 0! = 24 - 24 + 12 - 4 + 1 = 9$$

Write a function

```
function be = bernEuler(n)
% Post:  Number of ways to put n letters all in the wrong n envelopes.
% Pre:  1<=n<=21
```

and use it to print a table that shows $B_1, \ldots, B_{12}$. Make use of the user-defined functions in this Chapter.

PROBLEM 4.12. The number ways a set of n objects can be partitioned into m nonempty subsets is given by

$$\sigma_n^{(m)} = \sum_{j=1}^{m} \frac{(-1)^{m-j} j^n}{(m-j)!j!},$$

$1 \le m \le n$. For example,

$$\sigma_4^{(2)} = \frac{(-1)^{2-1} 1^4}{(2-1)!1!} + \frac{(-1)^{2-2} 2^4}{(2-2)!2!} = -1 + 8 = 7.$$

Thus, there are 7 ways to partition a 4-element set like $\{a, b, c, d\}$ into two non-empty subsets:

1 : {a},{b,c,d}
2 : {b},{a,c,d}
3 : {c}, {a,b,d}
4 : {d},{a,b,c}
5 : {a,b},{c,d}
6 : {a,c},{b,d}
7 : {a,d},{b,c}

Note that $\sigma_n^{(1)} = \sigma_n^{(n)} = 1$. Print a table with 10 lines. On line n should be printed the numbers $\sigma_n^{(1)}, \sigma_n^{(2)}, \ldots, \sigma_n^{(n)}$. Make use of the user-defined functions in this Chapter.

PROBLEM 4.13. If j and k are nonnegative integers that satisfy $j + k \le n$, then the coefficient of $x^k y^j z^{n-j-k}$ in $(x + y + z)^n$ is given by the *trinomial coefficient*

$$T(n, j, k) = \binom{n}{j} \binom{n-j}{k}$$

Write a function `triCoeff(n,j,k)` that computes $T(n, j, k)$ and use it to print a list of all trinomial coefficients of the form $T(10, j, k)$ where $0 \le j \le k$ and $j + k \le 10$. Make use of `binCoeff` and other user-defined functions in this Chapter.

PROBLEM 4.14. Modify the fragment

```
for a= 1:4
   for b= 1:4
```

```
      for c= 1:4
         for d= 1:4
            fprintf('%1d%1d%1d%1d\n', a, b, c, d)
         end
      end
   end
end
```

so that it prints a list of all possible permutations of the digits 1,2,3, and 4, i.e.,

```
1234   1243   1324   1342   1423   1432
2134   2143   2314   2341   2413   2431
3124   3142   3214   3241   3412   3421
4123   4132   4213   4231   4312   4321
```

(The order of the 24 numbers in the list is not important.)