Section 12

Strings, Time, Base Conversion and Bit Operations

12.1 Character Strings

While Matlab is primarily intended for number crunching, there are times when it is desirable to manipulate text, as is needed in placing labels and titles on plots. In Matlab, text is referred to as character strings or simply strings.

String Construction

Character strings in Matlab are special numerical arrays of ASCII values that are displayed as their character string representation. For example:

```
>> text = 'This is a character string'
text =
This is a character string
>> size(text)
ans =
     1    26
>> whos
  Name       Size         Bytes  Class

  ans        1x2             16  double array
  text       1x26            52  char array

Grand total is 28 elements using 68 bytes
```

ASCII Codes

Each character in a string is one element in an array that requires two bytes per character for storage, using the ASCII code. This differs from the eight bytes per element required for numerical or double arrays. The ASCII codes for the letters ‘A’ to ‘Z’ are the consecutive integers from 65 to 90, while the codes for ‘a’ to ‘z’ are 97 to 122. The function `abs` returns the ASCII codes for a string.

```
>> text = 'This is a character string'
text =
This is a character string
>> d = abs(text)
d =
  Columns 1 through 12
    84   104   105   115    32   105   115    32    97    32    99   104
  Columns 13 through 24
    97   114    97    99   116   101   114    32   115   116   114   105
  Columns 25 through 26
   110   103
```

The function `char` performs the inverse transformation, from ASCII codes to a string:

```
>> char(d)
ans =
This is a character string
```

The relationship between strings and their ASCII codes allow you to do the following:

```
>> alpha = abs('a'):abs('z');
>> disp(char(alpha))
abcdefghijklmnopqrstuvwxyz
```

Strings are Arrays

Since strings are arrays, they can be manipulated with array manipulation tools:

```
>> text = 'This is a character string';
>> u = text(11:19)
u =
character
```

As with matrices, character strings can have multiple rows, **but each row must have an equal number of columns.** Therefore, blanks are explicitly required to make all rows the same length. For example:

```
>> v = ['Character strings having more than'
        'one row must have the same number '
        'of columns - just like matrices   ']
v =
Character strings having more than
one row must have the same number
of columns - just like matrices
>> size(v)
ans =
     3    34
```

Concatenation of Strings

Because strings are arrays, they may be *concatenated* (joined) with square brackets. For example:

```
>> today = 'May';
>> today = [today, ' 18']
today =
May 18
```

String Conversions

`char(x)`	Converts the array `x` that contains positive integers representing character codes into a character array (the first 127 codes are ASCII). The result for any elements of `x` outside the range from 0 to 65535 is not defined.
`int2str(x)`	Rounds the elements of the matrix `x` to integers and converts the result into a string matrix.
`num2str(x)`	Converts the matrix `x` into a string representation with about 4 digits and an exponent if required. This is useful for labeling plots with the `title`, `xlabel`, `ylabel`, and `text` commands.
`str2num(s)`	Converts the string `s`, which should be an ASCII character representation of a numeric value, to numeric representation. The string may contain digits, a decimal point, a leading + or - sign, an 'e' preceeding a power of 10 scale factor, and an 'i' for a complex unit.

The `num2str` function can be used to convert numerical results into strings for use in formating displayed results with `disp`. For example, consider the following portion of a script:

```
tg = 2.2774;
xg = 144.6364;
disp(['time of flight:  ' num2str(tg) ' s'])
disp(['distance traveled  : ' num2str(xg) ' ft'])
```

The arguments for each of the `disp` commands are vectors of strings, with the first element being a label, the second being a number converted to a string, and the third being the units of the quantity. The label, the numerical results, and the units are displayed on one line, which was not possible with other forms of the use of `disp`:

```
time of flight:  2.2774 s
distance traveled:  144.6364 ft
```

String Functions

`blanks(n)`	Returns a string of `n` blanks. Used with `disp`, eg. `disp (['xxx' blanks(20) 'yyy'])`. `disp(blanks(n)')` moves the cursor down `n` lines.
`deblank(s)`	Removes trailing blanks from string `s`.
`eval(s)`	Execute the string `s` as a MATLAB expression or statement.
`eval(s1,s2)`	Provides the ability to catch errors. Executes string `s1` and returns if the operation was successful. If the operation generates an error, string `s2` is evaluated before returning. This can be thought of as `eval('try','catch')`.
`findstr(s1,s2)`	Find one string within another. Returns the starting indices of any occurrences of the shorter of the two strings in the longer.
`ischar(s)`	Returns 1 if `s` is a character array and 0 otherwise.
`isletter(s)`	Returns 1 for each element of character array `s` containing letters of the alphabet and 0 otherwise.
`isspace(s)`	Returns 1 for each element of character `s` containing white space characters and 0 otherwise. White space characters are spaces, newlines, carriage returns, tabs, vertical tabs, and formfeeds.
`lasterr`	Returns a string containing the last error message issued. `lasterr` is usually used in conjunction with the two argument form of `eval`: `eval('try','catch')`. The 'catch' action can examine the `lasterr` string to determine the cause of the error and take appropriate action.
`lower(s)`	Converts any uppercase characters in string `s` to the corresponding lowercase character and leaves all other characters unchanged.
`strcat(s1,s2,s3,...)`	Horizontally concatenates corresponding rows of the character arrays `s1`, `s2`, `s3` etc. The trailing padding is ignored. All the inputs must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array.
`strcmp(s1,s2)`	Returns 1 if strings `s1` and `s2` are identical and 0 otherwise.
`strjust(s)`	Returns a right justified version of the character array `s`.
`strmatch(str,strs)`	Searches through the rows of the character array of strings `strs` to find strings that begin with string `str`, returning the matching row indices.
`strncmp(s1,s2,n)`	Returns 1 if the first `n` characters of the strings `s1` and `s2` are identical and 0 otherwise.
`strrep(s1,s2,s3)`	Replaces all occurrences of the string `s2` in string `s1` with the string `s3`. The new string is returned.
`upper(s)`	Converts any lower case characters in `s` to the corresponding upper case character and leaves all other characters unchanged.

12.2 Time Computations

Current Date and Time

Three formats are supported for dates:

`clock`	Returns a six element date vector vector containing the current time and date in decimal form: `[year month day hour minute seconds]`. The first five elements are integers. The seconds element is accurate to several digits beyond the decimal point.
`date`	Returns a string containing the date in dd-mmm-yyyy format.
`now`	Returns the current date and time as a serial date number.

Examples:

```
>> t = clock
t =
       1998           6          10          10          18       59.57
>> date
ans =
10-Jun-1998
>> format long
>> d = now
d =
    7.299164376449074e+005
```

The date number can be converted to a string with the `datestr` function:

`datestr(d,dateform)`: Converts a data number `d` (such as that returned by `now`) into a date string. The string is formatted according to the format number `dateform` (see table below). By default, `dateform` is 1, 16, or 0 depending on whether `d` contains dates, times or both.

`dateform`	Date format	Example
0	`'dd-mmm-yyyy HH:MM:SS'`	`01-Mar-1995 15:45:17`
2	`'mm/dd/yy'`	`03/01/95`
3	`'mmm'`	`Mar`
4	`'m'`	`M`
5	`'mm'`	`3`
6	`'mm/dd'`	`03/01`
7	`'dd'`	`1`
8	`'ddd'`	`Wed`
9	`'d'`	`W`
10	`'yyyy'`	`1995`
11	`'yy'`	`95`
12	`'mmmyy'`	`Mar95`
13	`'HH:MM:SS'`	`15:45:17`
14	`'HH:MM:SS PM'`	`3:45:17 PM`
15	`'HH:MM'`	`15:45`
16	`'HH:MM PM'`	`3:45 PM`
17	`'QQ-YY'`	`Q1-96`
18	`'QQ'`	`Q1`

Examples:

```
>> ds = datestr(d)
ds =
10-Jun-1998 10:30:13
>> datestr(d,14)
ans =
10:30:13 AM
```

The function `datenum` is used to compute a date number. It has three forms:

`datenum(s)`: Returns the date number corresponding to the date string `s`.

`datenum(year,month,day)`: Returns the date number corresponding to the specified `year`, `month`, and `day`.

`datenum(year,month,day,hour minute,second)`: Returns the date number corresponding to the specified `year`, `month`, `day`, `hour`, `minute`, and `second`.

Examples:

```
>> datenum(ds)
ans =
    7.299164376504630e+005
>> datenum(1998,6,13)
ans =
      729919
```

```
>> datenum(1998,6,16,10,30,00)
ans =
    7.299224375000000e+005
```

Date Functions

The day of the week may be found from a date string or a date number using `weekday`, using the convention that Sunday = 1 and Saturday = 7.

```
>> [d s] = weekday('9/9/90')
d =
     1
s =
Sun
```

A calendar can be generated for a desired month, for display in the *Command* window or to be placed in a 6-by-7 array.

```
>> calendar('9/9/90')
                   Sep 1990
     S      M     Tu      W     Th      F      S
     0      0      0      0      0      0      1
     2      3      4      5      6      7      8
     9     10     11     12     13     14     15
    16     17     18     19     20     21     22
    23     24     25     26     27     28     29
    30      0      0      0      0      0      0
>> a = calendar(1978,6)
a =
     0      0      0      0      1      2      3
     4      5      6      7      8      9     10
    11     12     13     14     15     16     17
    18     19     20     21     22     23     24
    25     26     27     28     29     30      0
     0      0      0      0      0      0      0
```

Timing Functions

The functions `tic` and `toc` are used to time a sequence of commands. `tic` starts the timer; `toc` stops the timer and displays the elapsed time in seconds.

Example:

```
tic
```

```
for t=1:5000
    y(t)=sin(2*pi*t/10);
end
toc
```

Executing:

```
elapsed_time =
    4.5100
```

`cputime` returns the amount of central processing unit (CPU) time in seconds since the current session was started. Computing processing times at various points in a script and taking differences can be used to determine the CPU time required for segments of the script, possibly locating portions of the code that could be rewritten to decrease the total computation time.

12.3 Base Conversions and Bit Operations

Base Conversion

MATLAB provides functions for converting decimal numbers to other bases in the form of character strings. These conversion functions include:

`dec2bin(d)`	Returns the binary representation of `d` as a string. `d` must be a non-negative integer smaller than 2^{52}.
`dec2bin(d,N)`	Produces a binary representation with at least `N` bits.
`bin2dec(b)`	Interprets the binary string `b` and returns the equivalent decimal number.
`dec2hex(d)`	Returns the hexadecimal representation of decimal integer `d` as a string. `d` must be a non-negative integer smaller than 2^{52}.
`hex2dec(h)`	Interprets the hexadecimal string `h` and returns the equivalent decimal number. If `h` is a character array, each row is interpreted as a hexadecimal string.
`dec2base(d,b)`	Returns the representation of `d` as a string in base `b`. `d` must be a non-negative integer smaller than 2^{52} and `b` must be an integer between 2 and 36.
`dec2base(d,b,N)`	Produces a representation with at least `N` digits.

Examples:

```
>> a = dec2bin(18)     % find binary representation of 18
a =
10010
>> bin2dec(a)          % convert a back to decimal
ans =
    18
```

```
>> b = dec2hex(30)    % hex representation of 30
b =
1E
>> hex2dec(b)         % convert b back to decimal
ans =
    30
>> c = dec2base(30,4) % 30 in base 4
c =
132
>> base2dec(c,4)      % convert c back to decimal
ans =
    30
```

Bit Operations

MATLAB provides functions to implement logical operations on the individual bits of floating-point integers. The MATLAB bitwise functions operate on integers between 0 and `bitmax`, which is $2^{53} - 1 = 9.0072 \times 10^{15}$:

`c = bitand(a,b)`	Returns the bit-wise AND of the two arguments `a` and `b`.
`c = bitor(a,b)`	Returns the bit-wise OR of the two arguments `a` and `b`.
`c = bitxor(a,b)`	Returns the bit-wise exclusive OR of the two arguments `a` and `b`.
`c = bitcmp(a,N)`	Returns the bit-wise complement of `a` as an `N`-bit non-negative integer.
`c = bitset(a,bit,v)`	Sets the bit at position `bit` to the value `v`, where `v` must be either 0 or 1.
`c = bitget(a,bit)`	Returns the value of the bit at position `bit` in `a`. `a` must contain non-negative integers and `bit` must be a number between 1 and the number of bits in a floating point integer (52 for IEEE machines).
`c = bitshift(a,N)`	Returns the value of `a` shifted by `N` bits. If `N > 0`, this is same as a multiplication by 2^N (left shift). If `N < 0`, this is the same as a division by `2^(-N)` (right shift).

Examples of the use of these functions are best understood by displaying the results of each operation by `dec2bin`:

```
>> a = 6;
>> b = 10;
>> dec2bin(a,4)
ans =
0110
>> dec2bin(b,4)
ans =
1010
>> dec2bin(bitand(a,b),4)
ans =
0010
```

```
>> dec2bin(bitor(a,b),4)
ans =
1110
>> dec2bin(bitxor(a,b),4)
ans =
1100
>> dec2bin(bitcmp(a,4),4)
ans =
1001
>> dec2bin(bitset(a,4,1),4)
ans =
1110
>> bitget(a,2)
ans =
     1
>> dec2bin(bitshift(a,1))
ans =
1100
```

Section 13

Symbolic Processing

We have focused on the use of MATLAB to perform numerical operations, involving numerical data represented by double precision floating point numbers. We also given some consideration to the manipulation of text, represented by character strings. In this section, we introduce the use of MATLAB to perform *symbolic processing* to manipulate mathematical expressions, in much the way that we do with pencil and paper.

The objective of symbolic processing is to obtain what are known as *closed form* solutions, expressions that don't need to be iterated or updated in order to obtain answers. An understanding of these solutions often provides better physical and mathematical insight into the problem under investigation.

For more information, type `help symbolic` in MATLAB. A tutorial demonstration is available with the command `symintro`.

The following notes represent a short introduction to the symbolic processing capabilities of MATLAB.

13.1 Symbolic Expressions and Algebra

To introduce symbolic processing, first consider the handling of symbolic expressions and the manipulation of algebra.

Declaring Symbolic Variables and Constants

To enable symbolic processing, the variables and constants involved must first be declared as *symbolic objects*.

For example, to create the symbolic variables with names `x` and `y`:

```
>> syms x y
```

If x and y are to be assumed to be real variables, they are created with the command:

```
>> syms x y real
```

To declare symbolic constants, the sym function is used. Its argument is a string containing the name of a special variable, a numeric expression, or a function evaluation. It is used in an assignment statement which serves as a declaration of a symbolic variable for the assigned variable. Examples include:

```
>> pi = sym('pi');
>> delta = sym('1/10');
>> sqroot2 = sym('sqrt(2)');
```

If the symbolic constant pi is created this way, it replaces the special variable pi in the workspace. The advantage of using symbolic constants is that they maintain full accuracy until a numeric evaluation is required.

Symbolic variables and constants are represented by the data type *symbolic object*. For example, as displayed by the function whos for the symbolic variables and constants declared in the commands above:

```
>> whos
  Name           Size         Bytes  Class

  delta          1x1            132  sym object
  pi             1x1            128  sym object
  sqroot2        1x1            138  sym object
  x              1x1            126  sym object
  y              1x1            126  sym object

Grand total is 20 elements using 650 bytes
```

Symbolic Expressions

Symbolic variables can be used in expressions and as arguments of functions in much the same way as numeric variables have been used. The operators `+ - * / ^` and the built-in functions can also be used in the same way as they have been used in numeric calculations. For example:

```
>> syms s t A
>> f = s^2 + 4*s + 5
f =
s^2+4*s+5
>> g = s + 2
g =
s+2
```

```
>> h = f*g
h =
(s^2+4*s+5)*(s+2)
>> z = exp(-s*t)
z =
exp(-s*t)
>> y = A*exp(-s*t)
y =
A*exp(-s*t)
```

The symbolic variables `s`, `t`, and `A` are first declared, then used in symbolic expressions to create the symbolic variables `f`, `g`, `h`, `z`, and `y`. The displayed results show that the created variables remain as symbolic objects, rather than being evaluated numerically. Symbolic processing doesn't seem to obey the `format compact` command, as the displayed output is always double-spaced. These blank lines have been removed from these notes to conserve paper.

The variable `x` is the *default* independent variable, but as can be seen with the expressions above, other variables can be specified to be the independent variable. It is important to know which variable is the independent variable in an expression. The command to find the independent variable is:

`findsym(S)`	Finds the symbolic variables in a symbolic expression or matrix `S` by returning a string containing all of the symbolic variables appearing in `S`. The variables are returned in alphabetical order and are separated by commas. If no symbolic variables are found, `findsym` returns the empty string.

Examples based on the declarations and expressions in the examples above are:

```
>> findsym(f)
ans =
s
>> findsym(z)
ans =
A, s, t
```

The vector and matrix notation used in MATLAB also applies to symbolic variables. For example, the symbolic matrix `B` can be created with the commands:

```
>> n = 3;
>> syms x
>> B = x.^((0:n)'*(0:n))
B =
[   1,   1,   1,   1]
[   1,   x, x^2, x^3]
[   1, x^2, x^4, x^6]
[   1, x^3, x^6, x^9]
```

Manipulating Polynomial Expressions

In the examples above, symbolic variables were declared and were used in symbolic expressions to create polynomials. We now wish to manipulate these polynomial expressions algebraically.

The MATLAB commands for this purpose include:

`expand(S)`	Expands each element of a symbolic expression `S` as a product of its factors. `expand` is most often used on polynomials, but also expands trigonometric, exponential and logarithmic functions.
`factor(S)`	Factors each element of the symbolic matrix `S`.
`simplify(S)`	Simplifies each element of the symbolic matrix `S`.
`[n,d] = numden(S)`	Returns two symbolic expressions that represent the numerator expression `num` and the denominator expression `den` for the rational representation of the symbolic expression `S`.
`subs(S,old,new)`	Symbolic substitution, replacing symbolic variable `old` with symbolic variable `new` in the symbolic expression `S`.

These commands can be used to implement symbolic polynomial operations that were previously considered as numeric operations in Section 7.2 of these notes.

- **Addition:**

```
>> syms s
>> A = s^4 -3*s^3 -s +2;
>> B = 4*s^3 -2*s^2 +5*s -16;
>> C = A + B
C =
s^4+s^3+4*s-14-2*s^2
```

  Note that the result is correct, although it is not in form the we prefer, as the terms are not ordered in decreasing powers of `s`.

- **Scalar multiple:**

```
>> syms s
>> A = s^4 -3*s^3 -s +2;
>> C = 3*A
C =
3*s^4-9*s^3-3*s+6
```

- **Multiplication:**

```
>> syms s
>> A = s+2;
>> B = s+3;
>> C = A*B
```

```
C =
(s+2)*(s+3)
>> C = expand(C)
C =
s^2+5*s+6
```

- **Factoring:**

```
>> syms s
>> D = s^2 + 6*s + 9;
>> D = factor(D)
D =
(s+3)^2
>> P = s^3 - 2*s^2 -3*s + 10;
>> P = factor(P)
P =
(s+2)*(s^2-4*s+5)
```

- **Common denominator:** Consider the expression

$$H(s) = -\frac{1/6}{s+3} - \frac{1/2}{s+1} + \frac{2/3}{s}$$

This can be expressed as a ratio of polynomials by finding the common denominator for the three terms, as follows:

```
>> syms s
>> H = -(1/6)/(s+3) -(1/2)/(s+1) +(2/3)/s;
>> [N,D] = numden(H)
N =
s+2
D =
(s+3)*(s+1)*s
>> D = expand(D)
D =
s^3+4*s^2+3*s
```

Thus, $H(s)$ can be expressed in the form

$$H(s) = \frac{s+2}{s^3+4s^2+3s}$$

As a second example, consider

$$G(s) = s+4+\frac{2}{s+4}+\frac{3}{s+2}$$

Manipulating with Matlab:

```
>> syms s
>> G = s+4 + 2/(s+4) + 3/(s+2);
>> [N,D] = numden(G)
N =
s^3+10*s^2+37*s+48

D =
(s+4)*(s+2)
>> D = expand(D)
D =
s^2+6*s+8
```

Thus, $G(s)$ can also be expressed in the form:

$$G(s) = \frac{s^3 + 10s^2 + 37s + 48}{s^2 + 6s + 8}$$

In this example, $G(s)$ is an improper rational function.

- **Cancellation of terms**: For a ratio of polynomials, MATLAB can be applied to see if any terms cancel. For example

  $$H(s) = \frac{s^3 + 2s^2 + 5s + 10}{s^2 + 5}$$

  Applying MATLAB:

  ```
  >> syms s
  >> H = (s^3 +2*s^2 +5*s +10)/(s^2 + 5);
  >> H = simplify(H)
  H =
  s+2
  ```

  Factoring the denominator shows why the cancellation occurs:

  ```
  >> factor(s^3 +2*s^2 +5*s +10)
  ans =
  (s+2)*(s^2+5)
  ```

  Thus,

  $$H(s) = s + 2$$

- **Variable substitution:** Consider the ratio of polynomials

  $$H(s) = \frac{s + 3}{s^2 + 6s + 8}$$

  Define a second expression

  $$G(s) = H(s)|_{s=s+2}$$

  Evaluating $G(s)$ with MATLAB:

```
>> syms s
>> H = (s+3)/(s^2 +6*s + 8);
>> G = subs(H,s,s+2)
G =
(s+5)/((s+2)^2+6*s+20)
>> G = collect(G)
G =
(s+5)/(s^2+10*s+24)
```

Thus

$$G(s) = \frac{s+5}{s^2+10s+24}$$

Commands are also provided to convert between the numeric representation of a polynomial as the vector of its coefficients and the symbolic representation.

`sym2poly(P)`	Converts from a symbolic polynomial `P` to a row vector containing the polynomial coefficients.
`poly2sym(p)`	Converts from a polynomial coefficient vector `p` to a symbolic polynomial in the variable `x`. `poly2sym(p,v)` uses the symbolic the variable `v`.

For example, consider the polynomial

$$A(s) = s^3 + 4s^2 - 7s - 10$$

In MATLAB:

```
>> a = [1 4 -7 -10];
>> A = poly2sym(a,s)
A =
s^3+4*s^2-7*s-10
```

For the polynomial

$$B(s) = 4s^3 - 2s^2 + 5s - 16$$

```
>> syms s
>> B = 4*s^3 -2*s^2 +5*s -16;
>> b = sym2poly(B)
b =
     4    -2     5   -16
```

Forms of Expressions

As we have seen in some of the examples above, MATLAB does not always arrange expressions in the form that we would prefer. For example, MATLAB expresses results in the form `1/a*b`, while we would prefer `b/a`. For example:

```
>> syms s
>> H = s^2 +6*s + 8;
>> G = -H/3
G =
-1/3*s^2-2*s-8/3
```

The result is $G = -(1/3)s^2 - 2s - 8/3$, while we would prefer $G = -(s^2 + 6s + 8)/3$.

13.2 Manipulating Trigonometric Expressions

Trigonometric expressions can also be manipulated symbolically in MATLAB, primarily with the use of the `expand` function. For example:

```
>> syms theta phi
>> A = sin(theta + phi)
A =
sin(theta+phi)
>> A = expand(A)
A =
sin(theta)*cos(phi)+cos(theta)*sin(phi)
>> B = cos(2*theta)
B =
cos(2*theta)
>> B = expand(B)
B =
2*cos(theta)^2-1
>> C = 6*((sin(theta))^2 + (cos(theta))^2)
C =
6*sin(theta)^2+6*cos(theta)^2
>> C = expand(C)
C =
6*sin(theta)^2+6*cos(theta)^2
```

Thus, MATLAB was able to apply trigonometric identities to expressions `A` and `B`, but it was not successful with `C`, as we know

$$C = 6(\sin^2(\theta) + \cos^2(\theta)) = 6$$

MATLAB can also manipulate expressions involving complex exponential functions. For example:

```
>> syms theta real
>> A = real(exp(j*theta))
A =
1/2*exp(i*theta)+1/2*exp(-i*theta)
```

```
>> A = simplify(A)
A =
cos(theta)
```

13.3 Evaluating and Plotting Symbolic Expressions

In many applications, we eventually want to obtain numerical results or a plot from a symbolic expression. The function `double` produces numerical results:

`double(S)`	Converts the symbolic matrix expression `S` to a matrix of double precision floating point numbers. `S` must not contain any symbolic variables, except possibly `eps`.

Since the symbolic expression cannot contain any symbolic variables, it is necessary to use `subs` to substitute numerical values for the symbolic variables prior to applying `double`. For example:

```
>> E = s^3 -14*s^2 +65*s -100;
>> F = subs(E,s,7.1)
F =
13671/1000
>> G = double(F)
G =
   13.6710
```

The symbolic form is `F` and the numeric quantity is `G`, as confirmed by the display from `whos`:

```
>> whos
  Name      Size         Bytes  Class

  E         1x1            162  sym object
  F         1x1            144  sym object
  G         1x1              8  double array
  s         1x1            126  sym object

Grand total is 34 elements using 440 bytes
```

Symbolic expressions can be plotted with the MATLAB function `ezplot`:

`ezplot(f)`	Plots a graph of `f(x)` where `f` is a string or a symbolic expression representing a mathematical expression involving a single symbolic variable, say `x`. The default range of the x-axis is $[-2\pi, 2\pi]$
`ezplot(f,xmin,xmax)`	Plots the graph using the specified x-range instead of the default range.

For example consider plotting the polynomial function

$$A(s) = s^3 + 4s^2 - 7s - 10$$

over the range $[-1, 3]$:

```
syms s
a = [1 4 -7 -10];
A = poly2sym(a,s)
ezplot(A,-1,3), ylabel('A(s)')
```

The resulting plot is shown in Figure 13.3. Note that the expression plotted is automatically placed at the top of the plot and that the axis label for the independent variable is automatically placed. A `ylabel` command was used to label the dependent variable.

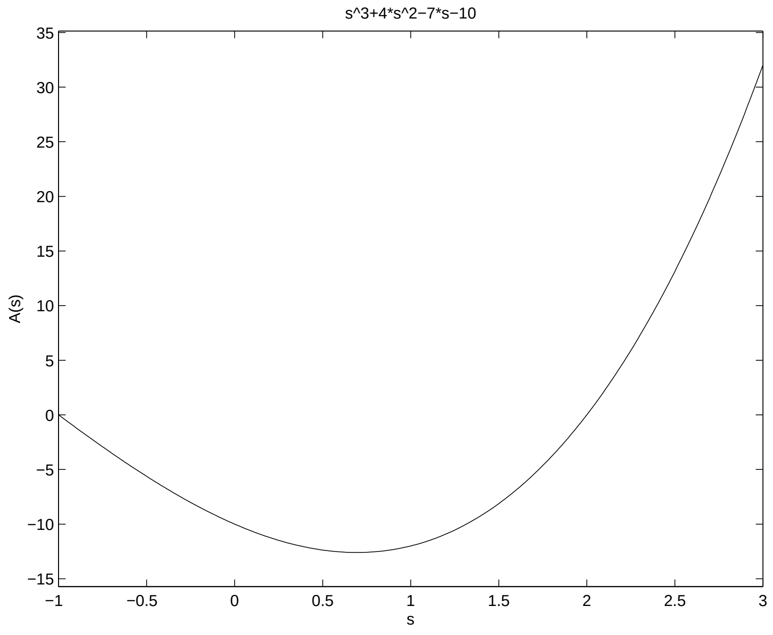

Figure 13.1: Plot of a polynomial function using `ezplot`

13.4 Solving Algebraic and Transcendental Equations

The symbolic math toolbox can be used to solve algebraic and transcendental equations, as well as systems of such equations. A *transcendental* equation is one that contains one or more transcendental functions, such as $\cos x$, e^x, or $\ln x$.

The function used in solving these equations is `solve`. There are several forms of `solve`, but only the following forms will be presented in these notes:

```
solve(E1, E2,...,EN)
solve(E1, E2,...,EN, var1, var2,...,varN)
```

where `E1, E2,...,EN` are the names of symbolic expressions and `var1, var2,..., varN` are variables in the expressions that have been declared to be symbolic. The solutions obtained are the roots of the expressions; that is, symbolic expressions for the variables under the conditions `E1 = 0`, `E2 = 0`, ... `EN = 0`.

For one equation and one variable, the resulting output solution is returned as a single symbolic variable.

For example:

```
>> syms s
>> E = s+2;
>> s = solve(E)
s =
-2
```

For `N` equations, the `N` solutions are returned as a symbolic vector.

For example:

```
>> syms s
>> D = s^2 +6*s +9;
>> s = solve(D)
s =
[ -3]
[ -3]
```

Thus, the solution is the symbolic representation of the repeated real roots of the quadratic polynomial, providing the same results as those obtained earlier in numeric representation using the function `roots`. Complex roots can also be obtained, as shown in the following example:

```
>> syms s
>> P = s^3 -2*s^2 -3*s + 10;
>> s = solve(P)
s =
[  -2]
[ 2+i]
[ 2-i]
```

Similar results can be obtained in the solution of transcendental equations. An example in trigonometry:

```
>> syms theta x z
>> E = z*cos(theta) - x;
>> theta = solve(E,theta)
theta =
acos(x/z)
```

For an example involving e^x, consider the solution to $e^{2x} + 4e^x = 32$:

```
>> syms x
>> E = exp(2*x) + 4*exp(x) -32;
>> x = solve(E)
x =
[ log(-8)]
[  log(4)]
>> log(-8)
ans =
   2.0794+ 3.1416i
>> log(4)
ans =
    1.3863
```

Note that two solutions are provided, with the numeric results showing that the first solution $\log(-8) = 2.0794 + 3.1416i$ is complex, while the second solution $\log(4) = 1.3863$ is real. The issue as to whether both of these solutions are meaningful would depend on the application that led to the original equation.

Equations containing periodic functions can have an infinite number of solutions. In such cases, `solve` restricts the search for solutions to the region near 0. For example, to solve the equation $\cos(2\theta) - \sin(\theta) = 0$:

```
>> E = cos(2*theta)-sin(theta);
>> solve(E)
ans =
[ -1/2*pi]
[  1/6*pi]
[  5/6*pi]
```

Example 13.1 *Positioning a robot arm*

Consider again the application to robot motion that was presented in Section 10.4. The robot arm has two joints and two links. The (x, y) coordinates of the hand are given by

$$x_1 = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$$

$$x_2 = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2)$$

where θ_1 and θ_2 are the joint angles and $L_1 = 4$ feet and $L_2 = 3$ feet are the link lengths. A part of the previous solution that was not determined was the joint angles needed to position the hand at a given set of coordinates. For the initial hand position of $(x, y) = (6.5, 0)$, the following commands determine the required angles:

```
>> syms theta1 theta2
>> E1 = 4*cos(theta1)+3*cos(theta1+theta2)-6.5;
>> E2 = 4*sin(theta1)+3*sin(theta1+theta2);
>> [theta1, theta2] = solve(E1,E2)
theta1 =
[  atan(9/197*55^(1/2))]
[ atan(-9/197*55^(1/2))]

theta2 =
[  -atan(3/23*55^(1/2))]
[ -atan(-3/23*55^(1/2))]
>> theta1 = double(theta1*(180/pi))
theta1 =
   18.7170
  -18.7170
>> theta2 = double(theta2*(180/pi))
theta2 =
  -44.0486
   44.0486
```

There are two solutions, given first in symbolic form, then converted into numeric form using `double`. The first is $\theta_1 = 18.717°$, $\theta_2 = -44.0486°$, which is the "elbow up" solution. The second is $\theta_1 = -18.717°$, $\theta_2 = 44.0486°$, the "elbow down" solution.

∎

13.5 Calculus

Symbolic expressions can be differentiated and integrated to obtain closed form results.

Differentiation

The `diff` function, when applied to a symbolic expression, provides a symbolic derivative.

`diff(E)`	Differentiates a symbolic expression `E` with respect to its free variable as determined by `findsym`.
`diff(E,v)`	Differentiates `E` with respect to symbolic variable `v`.
`diff(E,n)`	Differentiates `E` `n` times for positive integer `n`.
`diff(S,v,n)`	Differentiates `E` `n` times with respect to symbolic variable `v`.

Examples of derivatives of polynomial functions:

```
>> syms s n
>> p = s^3 + 4*s^2 -7*s -10;
>> d = diff(p)
d =
3*s^2+8*s-7
>> e = diff(p,2)
e =
6*s+8
>> f = diff(p,3)
f =
6
>> g = s^n;
>> h = diff(g)
h =
s^n*n/s
>> h = simplify(h)
h =
s^(n-1)*n
```

Examples of derivatives of transcendental functions:

```
>> syms x
>> f1 = log(x);
>> df1 = diff(f1)
df1 =
1/x
>> f2 = (cos(x))^2;
>> df2 = diff(f2)
df2 =
-2*cos(x)*sin(x)
>> f3 = sin(x^2);
>> df3 = diff(f3)
df3 =
2*cos(x^2)*x
>> df3 = simplify(df3)
df3 =
2*cos(x^2)*x
>> f4 = cos(2*x);
>> df4 = diff(f4)
df4 =
-2*sin(2*x)
>> f5 = exp(-(x^2)/2);
>> df5 = diff(f5)
df5 =
-x*exp(-1/2*x^2)
```

Min-Max Problems

The derivative can be used to find the maximum or minimum of a continuous function, say, $f(x)$, over an interval $a \leq x \leq b$. A *local* maximum or local minimum (one that does not occur at one of the boundaries $x = a$ or $x = b$) can occur only at a *critical point*, which is a point where either $df/dx = 0$ or df/dx does not exist.

Example 13.2 *Minimum cost tank design*

Consider again the tank design problem that was solved numerically in Example 7.1. In this problem, the tank radius is R, the height is H and the tank volume is such that

$$500 = \pi R^2 H + \frac{2}{3}\pi R^3$$

The cost of the tank, a function of surface area, is

$$C = 300(2\pi RH) + 400(2\pi R^2)$$

The problem is to solve for R and H providing the minimum cost tank providing the specified volume. The symbolic approach is to solve the volume equation for H as a function of R, express cost C symbolically, then differentiate C with respect to R and solve the resulting equation for R.

```
>> syms R H
>> V = pi*R^2*H + (2/3)*pi*R^3 -500;      % Equation for volume
>> H = solve(V,H)                         % Solve volume for height H
H =
-2/3*(pi*R^3-750)/pi/R^2
>> C = 300*(2*pi*R*H) + 400*(2*pi*R^2);   % Equation for cost
>> dCdR = diff(C,R)                       % Derivative of cost wrt R
dCdR =
400/R^2*(pi*R^3-750)+400*pi*R
>> Rmins = solve(dCdR,R)                  % Solve dC/dR for R: Rmin
Rmins =
[                               5/pi*3^(1/3)*(pi^2)^(1/3)]
[ -5/2/pi*3^(1/3)*(pi^2)^(1/3)+5/2*i*3^(5/6)/pi*(pi^2)^(1/3)]
[ -5/2/pi*3^(1/3)*(pi^2)^(1/3)-5/2*i*3^(5/6)/pi*(pi^2)^(1/3)]
>> Rmins = double(Rmins)
Rmin =
   4.9237
  -2.4619+ 4.2641i
  -2.4619- 4.2641i
>> Rmin = Rmins(1)
Rmin =
    4.9237
```

```
>> Hmin = double(subs(H,R,Rmin))
Hmin =
    3.2825
>> Cmin = double(subs(C,{R,H},{Rmin,Hmin}))
Cmin =
  9.1394e+004
```

Note that there are three symbolic solutions for R to provide minimum cost (`Rmins`). Converting these solutions to numeric quantities with `double`, we see that the second and third solutions are complex, which are not physically meaningful. Thus, we choose `Rmin` to be `Rmins(1)` and we compute `Hmin` and `Cmin` from this value. These symbolic results obtained here are more accurate than those determined previously in Example 7.1, as there has been no need to consider samples of R and H at a limited resolution. However, note that the results determined by the two methods are very close.

■

Integration

The `int` function, when applied to a symbolic expression, provides a symbolic integration.

`int(E)`	Indefinite integral of symbolic expression `E` with respect to its symbolic variable as defined by `findsym`. If `E` is a constant, the integral is with respect to `x`.
`int(E,v)`	Indefinite integral of `E` with respect to scalar symbolic variable `v`.
`int(E,a,b)`	Definite integral of `E` with respect to its symbolic variable from `a` to `b`, where `a` and `b` are each double or symbolic scalars.
`int(E,v,a,b)`	Definite integral of `E` with respect to `v` from `a` to `b`.

Examples of integrals of polynomials:

```
>> syms x n a b t
>> int(x^n)
ans =
x^(n+1)/(n+1)
>> int(x^3 +4*x^2 + 7*x + 10)
ans =
1/4*x^4+4/3*x^3+7/2*x^2+10*x
>> int(x,1,t)
ans =
1/2*t^2-1/2
>> int(x^3,a,b)
ans =
1/4*b^4-1/4*a^4
```

Examples of integrals of transcendental functions:

```
>> syms x
>> int(1/x)
ans =
log(x)
>> int(cos(x))
ans =
sin(x)
>> int(1/(1+x^2))
ans =
atan(x)
>> int(exp(-x^2))
ans =
1/2*pi^(1/2)*erf(x)
```

The last integral above introduces the error function `erf(x)` for each element of `x`, where `x` is real. The error function is defined as:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

13.6 Linear Algebra

Operations on symbolic matrices can be performed in much the same way as with numeric matrices.

The following are examples of matrix inverse, product, and determinant.

```
>> A = sym([2,1; 4,3])
A =
[ 2, 1]
[ 4, 3]
>> Ainv = inv(A)
Ainv =
[  3/2, -1/2]
[   -2,    1]
>> C = A*Ainv
C =
[ 1, 0]
[ 0, 1]
>> B = sym([1 3 0; -1 5 2; 1 2 1])
B =
[  1,  3,  0]
[ -1,  5,  2]
[  1,  2,  1]
>> detB = det(B)
detB =
10
```

Systems of linear equations can also be solved symbolically. Consider the following example that was previously solved numerically:

```
>> syms x
>> A = sym([3 2 -1; -1 3 2; 1 -1 -1])
A =
[  3,  2, -1]
[ -1,  3,  2]
[  1, -1, -1]
>> b = sym([10; 5; -1])
b =
[ 10]
[  5]
[ -1]
>> x = A\b
x =
[ -2]
[  5]
[ -6]
```

The results are the same as those obtained numerically. For this problem, there is little advantage to finding the result symbolically. However, solving a system of equations with respect to a parameter, the symbolic approach provides an advantage. Consider the following set of equations

$$\begin{aligned} 2x_1 - 3x_2 &= 3 \\ 5x_1 + cx_2 &= 19 \end{aligned}$$

To solve for x_1 and x_2 as functions of the parameter c:

```
>> syms c
>> A = sym([2 -3; 5 c]);
>> b = sym([3; 19]);
>> x = A\b
x =
[ 3*(19+c)/(2*c+15)]
[       23/(2*c+15)]
```