ENME202 Matlab

POLYNOMIALS

Polynominal Functions:

- polyval(p,x) Evaluate a polynomial array p at a given x value
- roots(p) Find the roots of a polynomial p
- conv(p1,p2) Multiply polynomials p1 and p2 (convolution)
- deconv(p1,p2) Divide polynomial p1 by p2 (deconvolution)
- poly(r) Return a polynomial from roots array r
- polyfit(x,y,s) Return a polynomial or order s fitting x,y data arrays

Polynomials can be represented in Matlab by arrays of coefficients, starting with the highest power of x.

For example, the following array can be used to represent p(x) = 2x^3 + 8x^2 + 12x + 8:

```
p = [2 8 12 8];
```

polyval() evaluates a polynomial at a specified point, i.e. for a specific numerical value of the independent variable:

```
polyval(p, 0)     % evaluate p(x) @ x=0
```

```
ans = 8
```

```
polyval(p, 2.15)  % evaluate p(x) @ x=2.15
```

```
ans = 90.6568
```

Just to verify Matlab's answer manually:

```
x = 2.15;
2*x^3 + 8*x^2 + 12*x + 8
```

```
ans = 90.6567
```

In the previous example, x was a single value, but polyval can also evaluate the polynomial across an entire array of points:

```
x = -3:.1:3;
polyval(p, x)     % returns an array with length(x) values
```

```
ans = 1×61
  -10.0000   -8.2980   -6.7840   -5.4460   -4.2720   -3.2500   -2.3680   -1.6140   -0.9760
```

Of course, we could do the same thing with regular Matlab array arithmetic:

```
2*x.^3 + 8*x.^2 + 12*x + 8
```

```
ans = 1×61
```

```
  -10.0000   -8.2980   -6.7840   -5.4460   -4.2720   -3.2500   -2.3680   -1.6140   -0.9760
```

The real value in Matlab's polynomial representation is access to a host of other polynomial functions as shown in the following discussion.

ROOTS

```
p = [1 0 0 0 -1]
```

```
p = 1×5
     1     0     0     0    -1
```

p is the array for the polynomial p(x) = x^4 - 1

Note that coefficients of "missing" powers of x are represented by zeros in the appropriate position of the coefficient array.

The **roots()** function will return the roots (zeros) of an arbitrary order polynomial:

```
r = roots(p)     % +/-1, +/-i (4 roots)
```

```
r = 4×1 complex
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Try with our earlier polynomial:

```
p = [2 8 12 8];  % 3rd order polynomial (4 elements)
r = roots(p)     % 3 roots: -2, -1+/-i
```

```
r = 3×1 complex
  -2.0000 + 0.0000i
  -1.0000 + 1.0000i
  -1.0000 - 1.0000i
```

Since the roots function finds roots numerically, evaluating the polynomial at each root produces a number close, but not equal, to 0 due to numerical error!

```
polyval(p, r(1))
```

```
ans = -8.8818e-15
```

POLYNOMIAL ARITHMETIC

Polynomial <u>addition</u> / <u>subtraction</u> is equivalent to adding/subtracting the coefficients of each array (but arrays must be same length!):

```
p1 = [1 2 3];      % p1(x) = x^2+2x+3
p2 = [3 1];        % p2(x) = 3x+1
% p3 = p1 + p2;      % Won't work -- arrays are different lengths
```

Need to "pad" the shorter coefficient array with leading zeros to make it the same length as the longer array in order to enable addition/subtraction:

```
p2 = [0 3 1];      % p2(x) = 0x^2 + 3x + 1
p3 = p1 + p2       % works!
```

```
p3 = 1×3
     1     5     4
```

Polynomial <u>multiplication</u>:

```
p1 = [3 4 8]             % p1(x) = 3x^2 + 4x + 8
```

```
p1 = 1×3
     3     4     8
```

```
p2 = [2 0 0 0 0 1]       % p2(x) = 2x^5 + 1
```

```
p2 = 1×6
     2     0     0     0     0     1
```

```
% p1 * p2                     % ERROR: Inner matrix dimensions must agree
```

Uh oh...what went wrong?

Polynomial multiplication is much more complicated than addition. Matlab has a special function called **conv()** (convolution) for polynomial multiplication:

```
conv(p1,p2)
```

```
ans = 1×8
     6     8    16     0     0     3     4     8
```

Note that no padding is needed for conv()

Above is the coefficient array for the product of (2x^5+1) * (3x^2+4x+8), which you can verify is correct by longhand calculation if you are particularly masochistic.

BUILDING A POLYNOMIAL FROM ITS ROOTS

```
p = [2 8 12 8];      % Our original polynomial
r = roots(p)         % --> -2, -1+/-i
```

```
r = 3×1 complex
  -2.0000 + 0.0000i
  -1.0000 + 1.0000i
  -1.0000 - 1.0000i
```

What if we were given the roots, and needed to find the corresponding polynomial?

We could use the roots to assemble the polynomial in its factored form by multiplying together all N of the first order factors (x-r(k)), where k = 1, 2, ..., N.

Let's start with the just two roots (since conv only takes two arguments):

```
p1 = conv([1 -r(1)],[1 -r(2)])
```

```
p1 = 1×3 complex
```

```
   1.0000 + 0.0000i   3.0000 - 1.0000i   2.0000 - 2.0000i
```

Then multiply the third polynomial by the result of the first product stored in p1:

```
p2 = conv(p1,[1 -r(3)])         % ans = [1 4 6 4]
```

```
p2 = 1×4 complex
   1.0000 + 0.0000i   4.0000 + 0.0000i   6.0000 + 0.0000i   4.0000 + 0.0000i
```

Check against original polynomial:

```
p    % [2 8 12 8] = 2 * p2
```

```
p = 1×4
     2     8    12     8
```

Note that the original polynomial isn't "monic", so we need to multiply the product of the factors by the coefficient of the highest power of x to recover the original polynomial:

```
2 * p2    % ans = [2 8 12 8]
```

```
ans = 1×4 complex
   2.0000 + 0.0000i   8.0000 + 0.0000i  12.0000 + 0.0000i   8.0000 + 0.0000i
```

We could also nest multiple conv() functions to accomplish the above in a single line:

```
2 * conv( conv([1 -r(1)], [1 -r(2)]), [1 -r(3)])
```

```
ans = 1×4 complex
   2.0000 + 0.0000i   8.0000 + 0.0000i  12.0000 + 0.0000i   8.0000 + 0.0000i
```

Ok, well that was a lot of work!! Now, let's do it the easy way using **poly()**.

poly() is a built-in command to expand a polynomial as the product of its first-order factors:

```
pp = poly(r)    % pp = [1 4 6 4]
```

```
pp = 1×4
    1.0000    4.0000    6.0000    4.0000
```

Same (monic) result as we got from conv above, but with a single function call. Very convenient!

POLYNOMIAL CURVE FITTING

Matlab can create a polynomial to fit a data set based on a least squares algorithm. Before investigating this, let's talk about saving and loading Matlab data files using the **save()** and **load()** functions:

- **save(filename)**   --> save current workspace variables to the named file
- **load(filename)**   --> load vaiables from named file to current workspace

Let's load two array variables from a file called sunspots.txt. This is a text file containing two columns: column 1 = observation year, and column 2 = average monthly number of sunspots observed.

```
load('sunspots.txt', '-ascii')      % 'ascii' or '-ascii' will work
```

Note that **sunspots.txt** must in the Matlab file path. You can add a new directory to your path via **HOME ➡ Set Path**.

The "-ascii" part of this function tells Matlab to load the data from a plain ASCII text file. After loading, there will be a new workspace variable with the same name as the file name (sunspots), which is an array containing the two data columns from the file:

```
sunspots
```

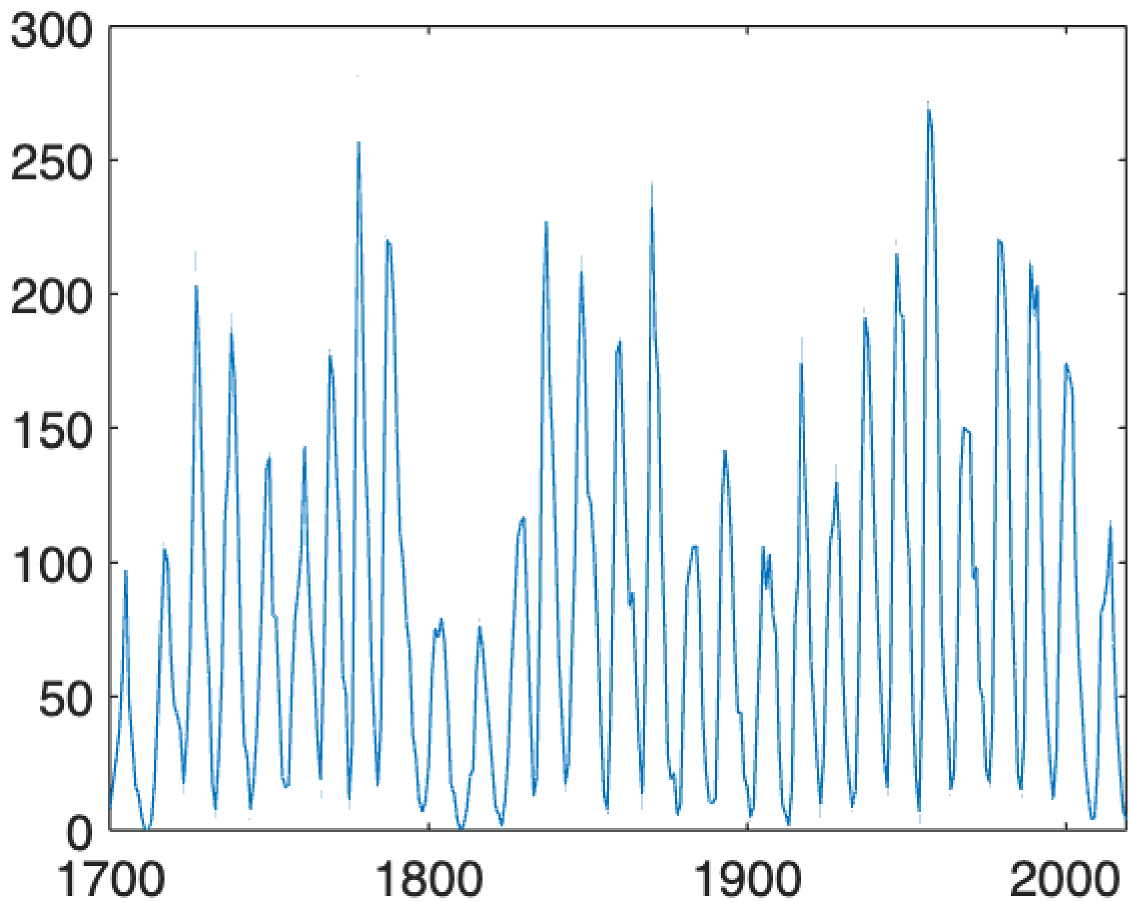
```
sunspots = 320×2
        1700           8
        1701          18
        1702          27
        1703          38
        1704          60
        1705          97
        1706          48
        1707          33
        1708          17
        1709          13
```

sunspots = [

Break 'sunspots' up into two separate variables for easier manipulation:

```
year = sunspots(:,1);   % slice 1st column
spots = sunspots(:,2);  % slice 2nd column

plot(year, spots)
```



Let's look only at the data since 1980. An easy way to do this is with the find() function:

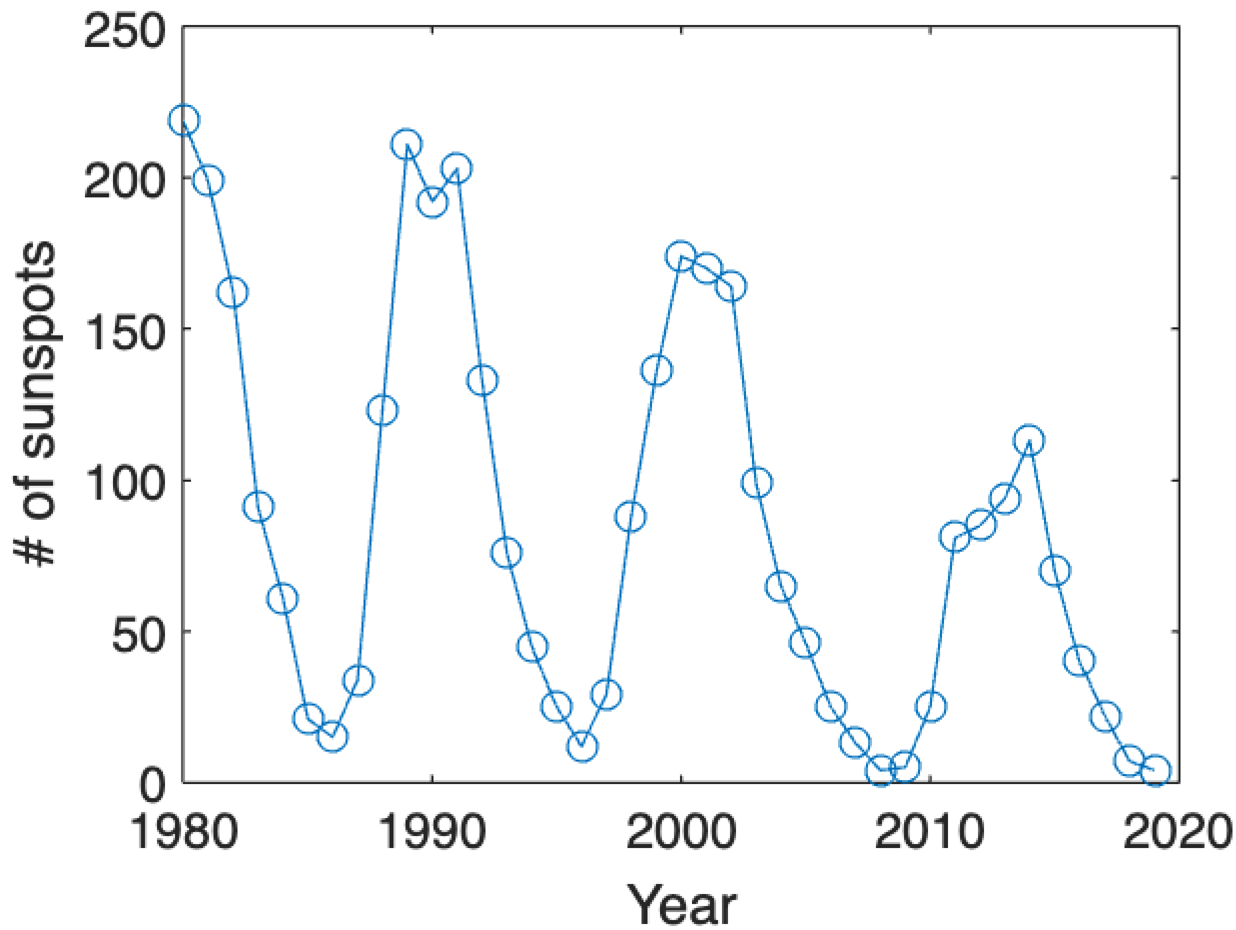
```
idx = find(year==1980);   % returns a single index
year = year(idx:end);
spots = spots(idx:end);
```

...or an alternate version of find():

```
indices = find(year>=1980);   % returns an array of indices
year = year(indices);
spots = spots(indices);
```

Plot the result:

```
plot(year, spots, 'o-')
xlabel('Year')
ylabel('# of sunspots')
```



Now let's fit a polynominal curve to the data. The **polyfit(x,y,s)** function takes 3 arguments:

1. x = x coordinates of data
2. y = y coordinates of dat
3. s = order of the polynomial to fit

The function returns an **array of polynomial coefficients** for the polynomial that best fits the given data by minimimzing the square of the error between the polynomial and data values (least squares method).

First, let's convert the year to a relative year (since 1980). The reason for this will be explained shortly:

```
delta_year = year - min(year);
pp = polyfit(delta_year, spots, 3);
```

Recall that polyval() converts a polynomial to an array of y-axis values for a given array of x-values. Let's use this to evaluate the polynomial against the original x data coordinates to see how well the function fits the data:

```
% First, replot the raw data:
plot(year, spots, 'o-')
xlabel('Year')
ylabel('# of sunspots')

hold on
```

```
fit = polyval(pp, delta_year);
plot(year, fit, 'r-')
```

Not very well here for a third order poly, so try a higher order:

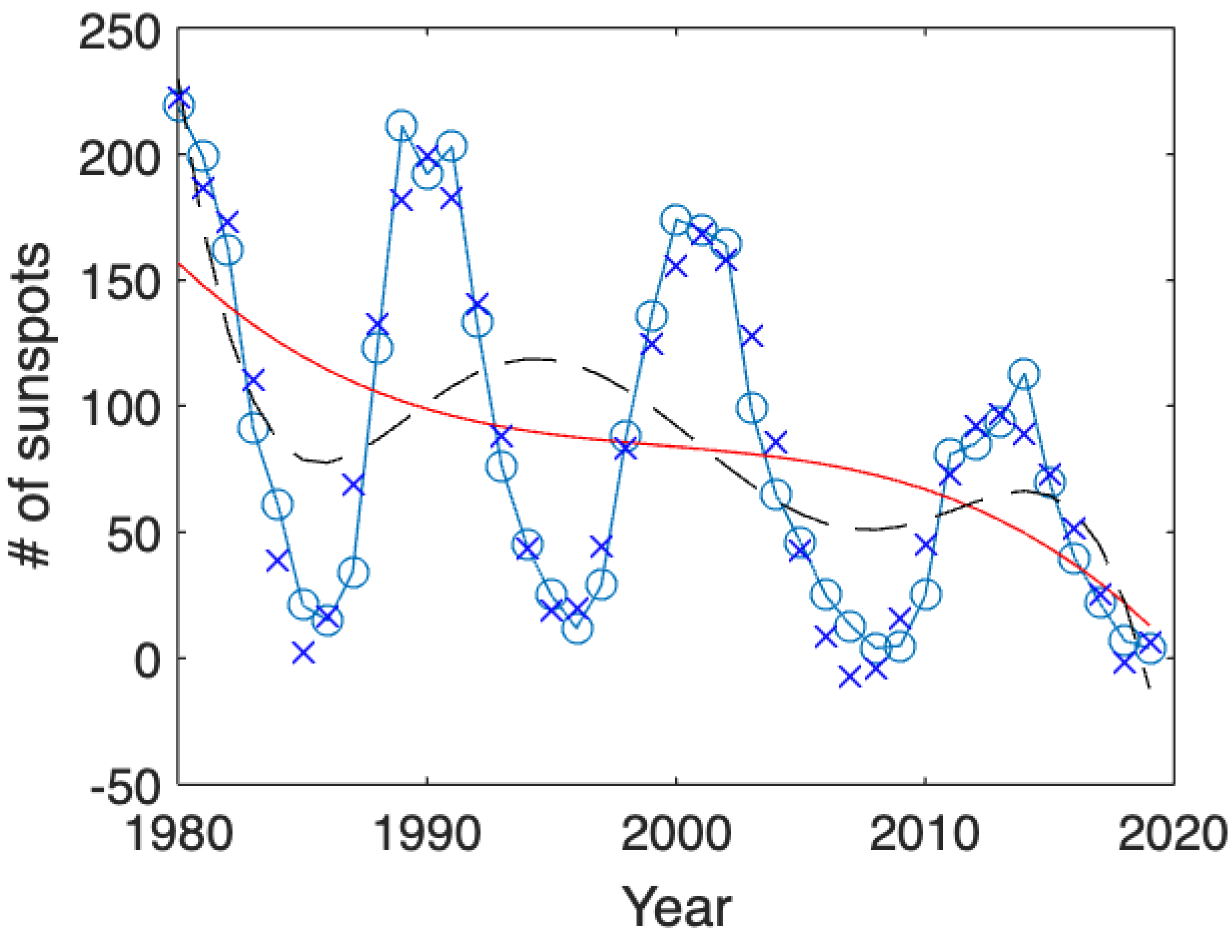
```
pp = polyfit(delta_year, spots, 5);
fit = polyval(pp, delta_year);
plot(year, fit, 'k--')
```

Even higher order:

```
pp = polyfit(delta_year, spots, 12);
```

```
Warning: Polynomial is badly conditioned. Add points with distinct X values, reduce the
degree of the polynomial, or try centering and scaling as described in HELP POLYFIT.
```

```
fit = polyval(pp, delta_year);
plot(year, fit, 'bx')
```



The resulting 12th order polynomial is "ill conditioned", meaning that in the calculation of the polynomial there was an operation that involved a near-singular matrix (a matrix close to not having an inverse). This is not necessarily a problem, but it is important to check the result to make sure the fit is acceptable.

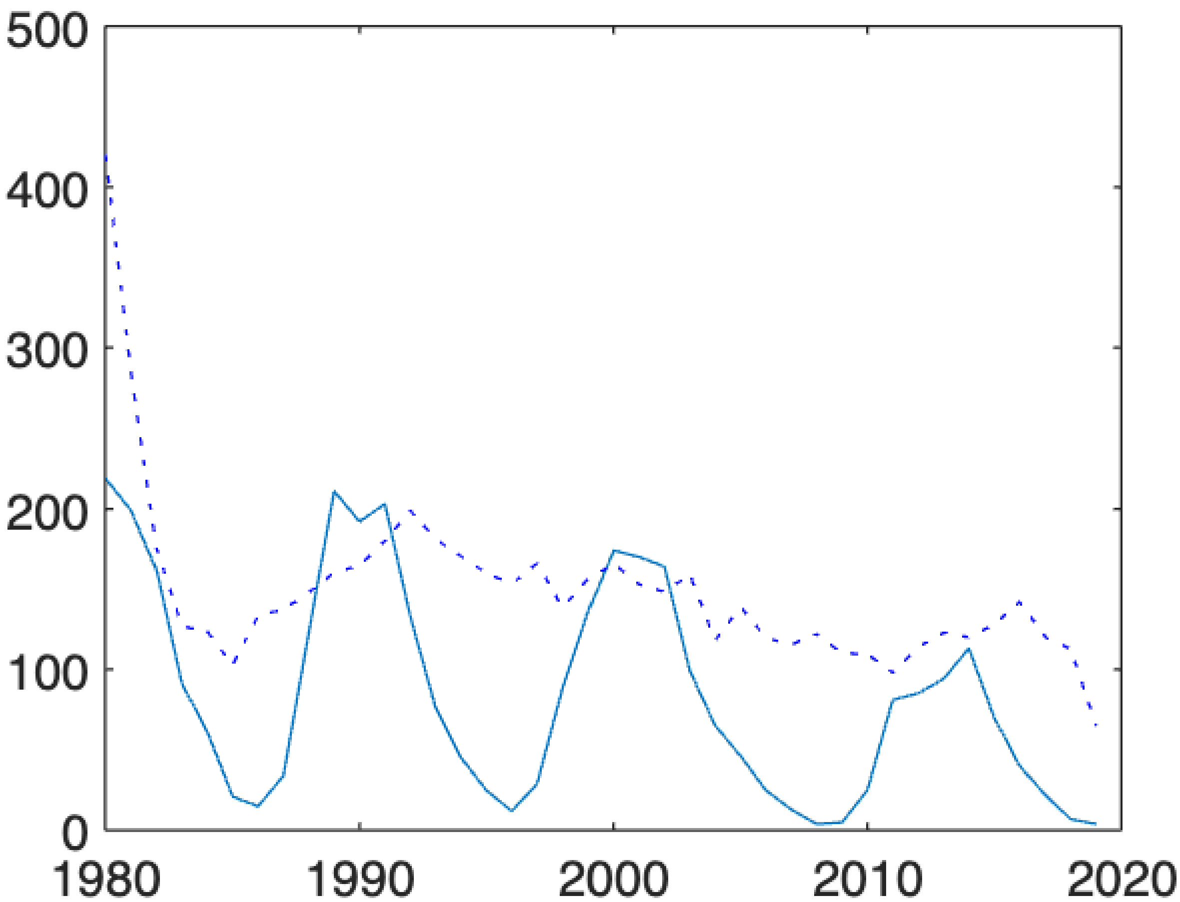
What would have happened if we used the *absolute* year instead of the relative year?

```
clf
```

```
pp = polyfit(year, spots, 12);
```

```
Warning: Polynomial is badly conditioned. Add points with distinct X values, reduce the
degree of the polynomial, or try centering and scaling as described in HELP POLYFIT.
```

```
plot(year,spots)
hold on
plot(year,polyval(pp,year),'b:')
```



What happened? The polynomial is trying to fit data far from the origin, without any intervening data to "guide" the fit, resulting in a poor fit.

POLYNOMIAL DIVISION

Like polynomial multiplication, Matlab can also do polynomial division using the **deconv()** function.

Note that in general:

p1(x)/p2(x) = q(x) + r(x)/p2(x)

q(x) is the **quotient** polynomial

r(x) is the **remainder** polynomial

```
p1 = [2 6 8 4];          % p1(x) = 2x^3+6x^2+8x+4
p2 = [1 2 2];            % p2(x) = x^2+2x+2
[q,r] = deconv(p1,p2)
```

```
q = 1×2
     2     2
r = 1×4
     0     0     0     0
```

If you calculate p1/p2 by hand, you will find that

p1(x)/p2(x) = x+4 with remainder -2x-4

So in Matlab, q = [1 4] for this problem, and r = [-2 -4]

Given q,r we can reconstruct the original polynomial as:

p1(x) = q(x)*p2(x) + r(x)

Note that r is **automatically padded** to the correct length so that we can reconstruct the original (p1) polynomial from:

```
conv(q, p2) + r      % ans = [1 6 8 4]
```

```
ans = 1×4
     2     6     8     4
```

A different example:

```
p2 = [1 1];      % p2 = x + 1
[q,r] = deconv(p1, p2) % x^3+6x^2+8x+4 / x+1
```

```
q = 1×3
     2     4     4
r = 1×4
     0     0     0     0
```

```
conv(q, p2) + r        % same quotient as above but new remainder
```

```
ans = 1×4
     2     6     8     4
```

A note about Matlab data files:

The above year & spot data were loaded from a **ascii-format text file** (containing values separated by white space or commas). However, data can also be saved in a **.mat** file, containing a custom data format defined by Matlab. Selected variable values can be saved using the **save()** function. For example, to save variables v1 and v2 to datafile.mat:

save('datafile', 'v1', 'v2')

Alternately, the entire workspace can be saved from the menu via **HOME ➡ save workspace**.

Use the **load** command to load data from a .mat file, **ommiting the extension**:

load('datafile')

or

```
load mydata
```

Any variables saved in the data file will be loaded into the current workspace.

Remember: when loading plain ascii text data, give the full file name **with extension** in quotes, and add **'-ascii'** to the load command:

```
load('mydata.txt', '-ascii')
```