3 INTERPOLATION

Contents

Initialization

```
% reduce needless whitespace
format compact
% reduce irritations
more off
% start a diary
%diary lectureN.txt
```

INTERPOLATION.

Probably, you have already done interpolation before in other courses. In fact, if you are in ME Tools, you are doing it there right now. The next few sections will explain how you can do it much easier and better with Matlab.
As an *example* problem that requires interpolation, assume that we have placed a hot bar with its ends in contact with ice water. The temperature of the bar will then decay over time to 0 degrees Centigrade. We have measured the temperature of the center of the bar at 6 times spaced half a minute apart. Taking the first of these times as time zero, the measured data are:

```
time:          0      0.5    1      1.5    2      minutes
Temperature:   14.60  8.42   4.86   2.80   1.62   Centigrade
```

We will define Matlab arrays `timeMeasured` and `TempMeasured` as the six measured times and temperatures respectively.
(Note to some students. This is a lecture about interpolation, *not* heat conduction. You do not need to understand heat conduction to follow this lecture. All

you need to know that we want to interpolate the data above. If you want, you can think of them instead as tabulated values in a ME Tools table.)
Supposedly unknown to us, the exact temperature is given by

$$T_{\text{exact}} = 14.6\exp(-1.1t);$$

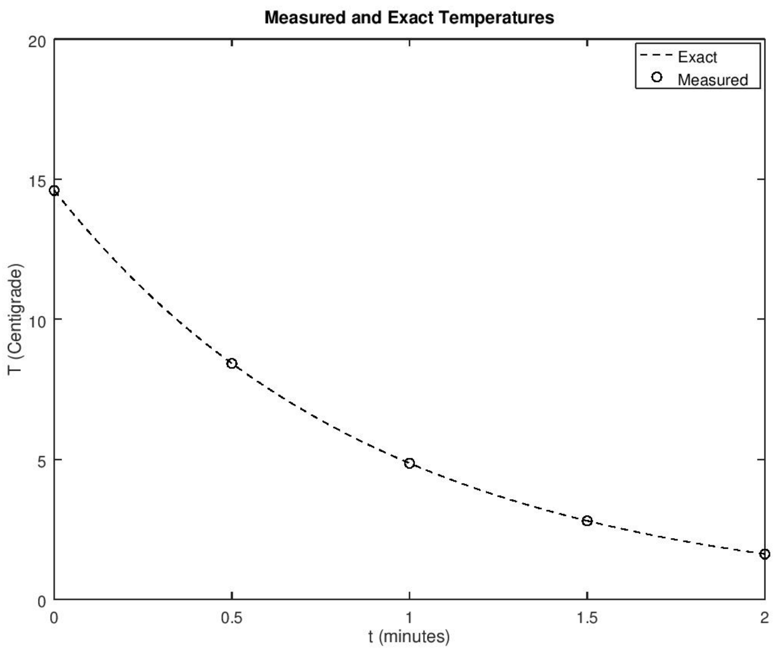
We will *pretend* that we only know the measured temperatures. So we have to interpolate using *only* those measured data. But *afterwards* we will cheat and evaluate the errors using the exact function above. Just to see how well we are really doing interpolating.
To make that easier, we will create a function `TempExactFun` to evaluate the exact temperature that we pretend not to know. Since the function is very simple and not intended for more general use, we do not need to create a function file for it. Instead we can define `TempExactFun` as a "handle" to an anonymous function.

```
% define timeMeasured and TempMeasured as given
timeMeasured=[   0     0.5     1     1.5     2]';
TempMeasured=[14.60  8.42  4.86  2.80  1.62]';

% make TempExactFun a handle to an anonymous function
TempExactFun = @(t) 14.6*exp(-1.1*t);
disp(' ')
```

Plot to understand the problem better

Let's plot the measured five values versus the exact solution that we pretend not to know. Note that to plot a function, you need to create a set of plot points. These are *different* from the measured points and just used for plotting.

```
% generate 100 time values between 0 and 2
timePlot=linspace(0,2,100);
% generate corresponding exact temperatures
TempExactPlot=TempExactFun(timePlot);

% create the plot, using circles for the measured points
plot(timePlot,TempExactPlot,'--k',...
     timeMeasured,TempMeasured,'ok')
legend('Exact','Measured')
title('Measured and Exact Temperatures')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
disp(' ')
```



Doing the interpolation

We would now like to be able to evaluate the temperature at times in between the measured five times. This is called "interpolation".
For example, let's assume that we want to know the temperature at time 0.7, which is in between measured times 0.5 and 1.
Matlab provides `interp1` or `spline` to find it.

```
% let's evaluate T at t = 0.7 using two different methods
time=0.7
TempLinear=interp1(timeMeasured,TempMeasured,time)
TempSpline=spline(timeMeasured,TempMeasured,time)

% two reasonable values, but which one is best???
TempExact=TempExactFun(time)
disp('For a nice smooth curve, spline interpolation is')
disp('much more accurate than linear interpolation!')
errLinear=abs(TempLinear-TempExact)
errSpline=abs(TempSpline-TempExact)
disp(' ')
```

```
time =  0.70000
TempLinear =  6.9960
```

```
TempSpline =   6.7513
TempExact =   6.7600
For a nice smooth curve, spline interpolation is
much more accurate than linear interpolation!
errLinear =   0.23601
errSpline =   0.0086708
```

Compare the interpolations

```
% find the interpolated values at the plot times
TempLinearPlot=...
    interp1(timeMeasured,TempMeasured,timePlot);
TempSplinePlot=...
    spline(timeMeasured,TempMeasured,timePlot);

% compare the interpolations in a plot
plot(timePlot,TempExactPlot,'--k',...
     timeMeasured,TempMeasured,'ok',...
     timePlot,TempLinearPlot,'r',...
     timePlot,TempSplinePlot,'b')
legend('Exact','Measured','Linear','Spline')
title('Linear and Spline Interpolation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% compare the _maximum_ deviations
errLinearPlot=max(abs(TempLinearPlot-TempExactPlot))
errSplinePlot=max(abs(TempSplinePlot-TempExactPlot))
disp('The spline is everywhere much more accurate.')
disp(' ')
```

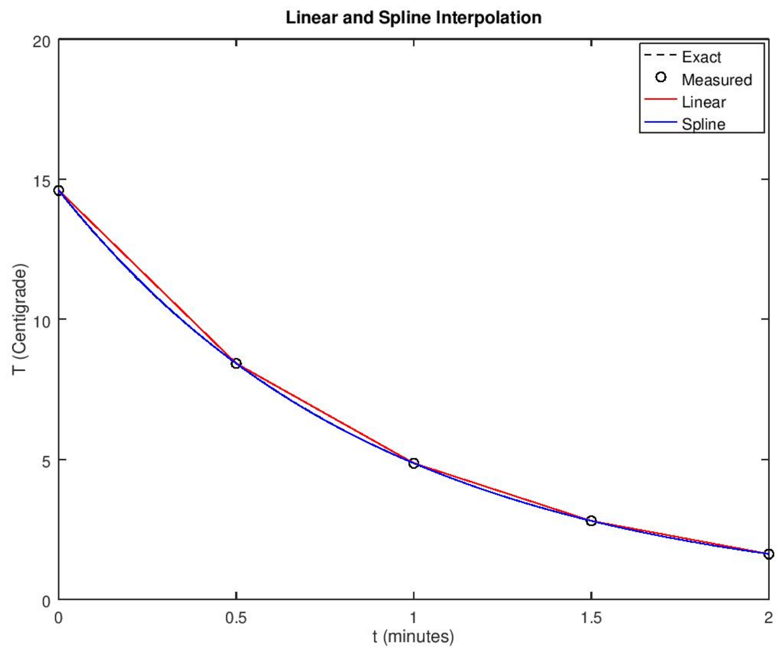
```
errLinearPlot =   0.42108
errSplinePlot =   0.017672
The spline is everywhere much more accurate.
```

Extrapolation

Suppose that the time at which we want to know the temperature is $t = 5$. This time is not inside the measured range from 0 to 2. If that happens, we talk about *extrapolation* instead of *interpolation*.

```
Extrapolation is much trickier than interpolation.
```

For that reason, `interp1` refuses to do it unless you specify an additional "extrap" parameter. Function `spline` will do it as is.




```
% evaluate the values at t = 5
time=5
TempExact=TempExactFun(time)
TempLinear=interp1(timeMeasured,TempMeasured,time,...
                   'linear','extrap')
TempSpline=spline(timeMeasured,TempMeasured,time)
disp('Extrapolation is usually bad news!')
disp(' ')

% Note that both linear and spline values are bad, and
% that the spline is much worse than linear.  But both
% values are useless.
```

```
time =   5
TempExact =   0.059667
TempLinear = -5.4600
TempSpline = -14.700
Extrapolation is usually bad news!
```

SKIP: More on spline interpolation

Often you would want your spline to satisfy end conditions. For example, you might want it to have given derivatives at the ends. Or be periodic. Given derivatives at the ends can be achieved using 'spline' if you add the desired two values to the function values list. For more complicated cases, consider function 'csape'.

NOISY DATA

What if the measured data have random errors? Suppose, for example, that the digital thermometer used to to measure the data only displays whole degrees C? Then the measured data:

```
Temperature: 14.60  8.42  4.86  2.80  1.62  Centigrade
```

become:

```
Temperature: 15  8  5  3  2  Centigrade
```

Then what happens to our interpolations?

```
% correct the measured data list
TempMeasured=[15 8 5 3 2]';

% interpolate again at t = 0.7
time=0.7
TempExact=TempExactFun(time)
TempLinear=interp1(timeMeasured,TempMeasured,time)
TempSpline=spline(timeMeasured,TempMeasured,time)
disp('Now the linear interpolation is actually better!');
errLinear=abs(TempLinear-TempExact)
errSpline=abs(TempSpline-TempExact)

% compare the interpolations in a plot
TempLinearPlot=...
    interp1(timeMeasured,TempMeasured,timePlot);
TempSplinePlot=...
    spline(timeMeasured,TempMeasured,timePlot);
plot(timePlot,TempExactPlot,'--k',...
     timeMeasured,TempMeasured,'ok',...
     timePlot,TempLinearPlot,'r',...
     timePlot,TempSplinePlot,'b')
legend('Exact','Measured','Linear','Spline')
title('Linear and Spline Interpolation, Noisy Data')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
```

```
% Because of the noise, the spline can be worse than
% linear.  The spline may also start oscilating if things
% get really bad.  Note the poor slope of the spline near
% time 2.  And examine your homework solutions.

% compare the _maximum_ deviations
errLinearPlot=max(abs(TempLinearPlot-TempExactPlot))
errSplinePlot=max(abs(TempSplinePlot-TempExactPlot))
disp('There is no longer a real difference in error.')

% The maximum deviations are practically speaking the
% same.  There is no longer a good reason to use spline
% interpolation instead of the simpler linear
% interpolation.
disp(' ')
```

```
time =   0.70000
TempExact =   6.7600
TempLinear =   6.8000
TempSpline =   6.5300
Now the linear interpolation is actually better!
errLinear =   0.040009
errSpline =   0.22999
errLinearPlot =   0.52550
errSplinePlot =   0.44453
There is no longer a real difference in error.
```

SAVING AND RELOADING

You can save all work space variables in a file `lecture4.mat` using the

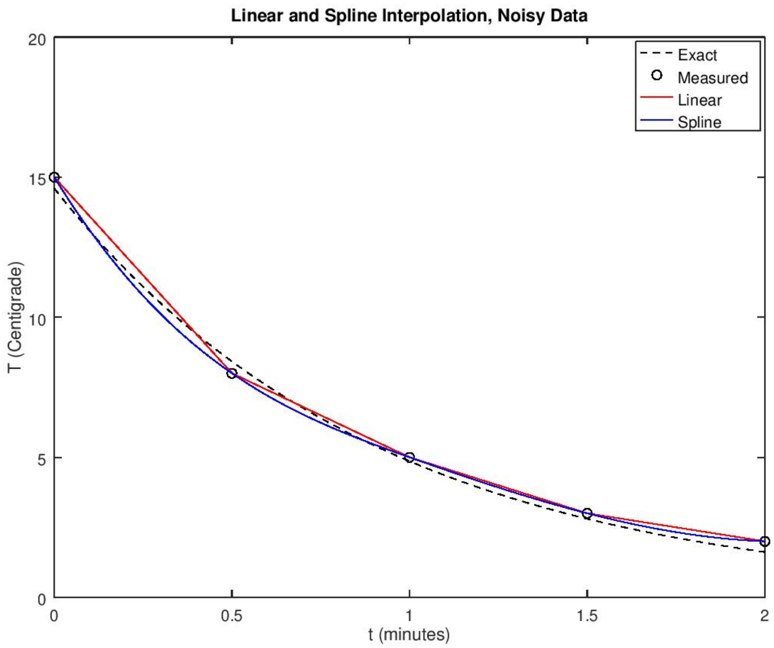
```
save lecture4
```

command. Then next time, you can resume where you left off using the

```
load lecture4
```

command.
Some things you may want to remember for future use: First, to save only a few variables, you could use the `save FILENAME VAR1 VAR2 ...` command. Second, to read in data from an Excel spreadsheet, use the `xlsread` command. To write data to an Excel sheet, use `writetable` or `xlswrite`. Use "cell arrays" if not all data is numerical.

```
% see what variables are defined
%who
% save them all in file lecture4.mat
```




```
%save lecture4

% let's test it works OK

% kill all variables in the work space
%clear
% check that they are gone (no response)
%who
% reload the variables from file lecture4.mat
%load lecture4
% check that they are back.
%who
```

CURVE FITTING

Functions `interp1` and `spline` reproduce the given measured data exactly. This was fine when the measured data were exact. However, the noisy measured data we are looking at now have *errors*. Functions `interp1` and `spline` will reproduce these *errors* exactly too. And that is bad news because of course we do not want these errors.

So if we want something more accurate than `interp1` and `spline`, we *must* drop the assumption that our interpolation reproduces all the measured data *exactly*. The interpolation we want should be *close* to the measured data, but it should not swing around wildly to go *exactly* through each measured point.
To prevent our interpolation from wildly swinging around, what we can do choose a relatively *simple* curve type. Then we can adjust that curve type to be *on average* as close as possible to the data points. This idea is called "curve fitting".
In particular, recall that the *exact* temperature curve is given by

$$T_{\text{exact}} = 14.6\exp(-1.1t);$$

However, we are assuming that we do not know that. And given only our noisy data, there is *no way* to figure out that the above is the exact temperature.
But suppose that we can guess (based on theoretical arguments not of importance here) that the desired temperature is of the form

$$T = A\exp(Bt);$$

Using that as the interpolating function, there is no possibility of wildly swinging about. And we can still choose values for the constants A and B that produce the best approximation to the measured data. This is sure to produce a better result than `interp1` and `spline`.
Of course, the devil is in the details. In particular, how are you going to find the best A and B? You could select A and B to make the curve go exactly through two 2 of the 5 measured temperatures. But which 2? If you are very lucky you could get a quite good approximation that way. But if you are unlucky, you would get unnecessarily big errors.
It is a much better idea to use *all* 5 measured data you have, and make the curve approximate them *on average* as well as it can. Typically, numerical analysist take "on average" to mean that they make the *average square error* as small as possible. There are both theoretical and practical reasons to do that:

1. Theoretically, in simple cases where the errors are truly random, this gives the best approximation possible (according to mathematical statistics).

2. Practically, the mathematics of making the average *square* error as small as possible is a lot simpler than other possibilities (like making the maximum error as small as possible).

We do not really need to worry about the latter anyway, as Matlab does that work for us. What we should get away with is that what we are going to do is popularly known as the "Method of Least Squares". (Though "Method of Least Average Square Error" would be more accurate.)

Fitting a line

Finding the best exponential approximation of the form

$$T = A\exp(Bt)$$

is actually somewhat messy.
So, for now, we will restrict ourself to simpler approximations. And the simplest approximation possible is surely by a straight line,

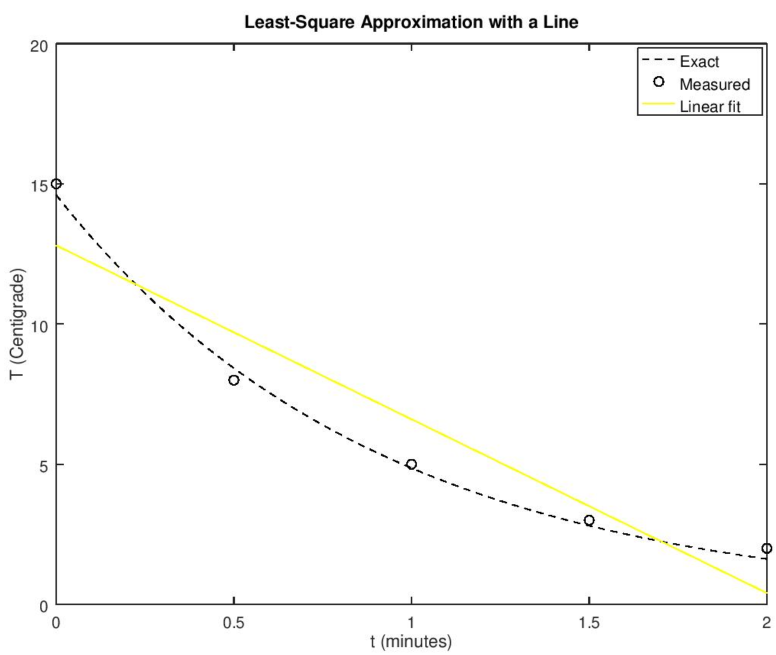
$$T = C_1 t + C_2;$$

(call it `TempLinFit` in Matlab).
If we settle for that as the interpolating function, Matlab can help us by finding the "best" (in the least square sense) values for the coefficients C_1 and C_2 for us. All *we* need to do is use a function called `polyfit` (for "fit a polynomial") on the measured data. (Note that the straight line relationship above is a polynomial of degree 1, since the highest power of x is 1.)
And having found the coefficients C_1 and C_2 of the polynomial with `polyfit`, we can use another Matlab function, `polyval` (for "find values of a polynomial"), to evaluate the polynomial at whatever times we want.
Note some more important terminology that you will frequently encounter in interpolation. In particular, the expression for T above is *linear* in the coefficients C_1 and C_2 to find. That is unlike for the exponential fit, where the coefficient B was *inside an exponential*, and that was then *multiplied* by A to boot. If the approximate expression is linear in terms of the unknown coefficients, like the straight line above, numerical analysists speak of "linear regression". Like "method of least squares", "linear regression" is another term you should try to remember.

```
% find the coefficients C1 and C2 of the fitted line
n=1;
CoefLinFit=polyfit(timeMeasured,TempMeasured,n)

% interpolate again at t = 0.7
time=0.7
TempExact=TempExactFun(time)
TempLinFit=polyval(CoefLinFit,time)
errLinFit=abs(TempLinFit-TempExact)
disp('OOPS!  That is horrible!')

% let's see the linear fit in a plot
TempLinFitPlot=polyval(CoefLinFit,timePlot);
plot(timePlot,TempExactPlot,'--k',...
      timeMeasured,TempMeasured,'ok',...
      timePlot,TempLinFitPlot,'y')
legend('Exact','Measured','Linear fit')
title('Least-Square Approximation with a Line')
```

```
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% print the error
errLinFitPlot=max(abs(TempLinFitPlot-TempExactPlot))
disp('That is horrible, but what do you expect?')
disp('Clearly no straight line could approximate')
disp('the exact curve in this example well.')
disp(' ')
```

```
CoefLinFit =
   -6.2000    12.8000
time =  0.70000
TempExact =  6.7600
TempLinFit =  8.4600
errLinFit =  1.7000
OOPS!  That is horrible!
errLinFitPlot =  1.8000
That is horrible, but what do you expect?
Clearly no straight line could approximate
the exact curve in this example well.
```



Fitting a parabola

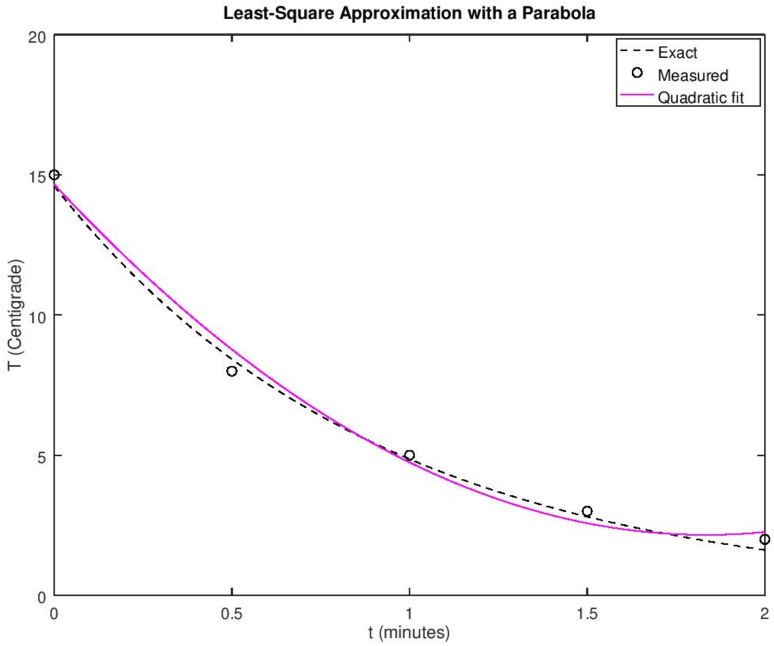
We can improve things quite a lot by approximating with a quadratic polynomial, i.e. a parabola,

$$T = C_1 t^2 + C_2 t + C_3;$$

instead of a straight line.
We will call this `TempParFit` (parabolic Temperature fit) in Matlab.

```
% find coefficients C1, C2, and C3
n=2;
CoefParFit=polyfit(timeMeasured,TempMeasured,n)

% interpolate again at t = 0.7
time=0.7
TempExact=TempExactFun(time)
TempParFit=polyval(CoefParFit,time)
disp('That is much better than the linear fit.')

% let's see the quadratic fit in a plot
TempParFitPlot=polyval(CoefParFit,timePlot);
plot(timePlot,TempExactPlot,'--k',...
     timeMeasured,TempMeasured,'ok',...
     timePlot,TempParFitPlot,'m')
legend('Exact','Measured','Quadratic fit')
title('Least-Square Approximation with a Parabola')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
disp('Not too bad.')

% print the error
errParFitPlot=max(abs(TempParFitPlot-TempExactPlot))
disp('But the maximum error, at t=2, is quite big.')
disp(' ')
```

```
CoefParFit =
    3.7143   -13.6286    14.6571
time =  0.70000
TempExact =  6.7600
TempParFit =  6.9371
That is much better than the linear fit.
Not too bad.
errParFitPlot =  0.63942
But the maximum error, at t=2, is quite big.
```



Fitting a quartic

Let's try fitting with a quartic,

$$T = C_1 t^4 + C_2 t^3 + C_3 t^2 + C_4 t + C_5;$$

Note however, that now we are no longer *fitting*, but *interpolating*. With 5 unknown coefficients, the quartic can go through all 5 measured data points. This is usually a very bad idea.
In this particular case, the results below are much better than I expected. Fitting curves with too many coefficients can give very bad results. In this case the only real problem is the slope at $t = 2$. It might have been much worse.

```
The general rule of thumb is:
  Do not interpolate a polynomial of degree more than
  about the square root of the number of data points
```

Since we have 5 data points and sqrt(5) is about 2, we should not fit a polynomial of a degree greater than 2.
Exceptions confirm the rule.

```
% find the 5 coefficients
n=4;
CoefQuartFit=polyfit(timeMeasured,TempMeasured,n)
```

```
% interpolate again at t = 0.7
time=0.7
TempExact=TempExactFun(time)
TempQuartFit=polyval(CoefQuartFit,time)
errQuartFit=abs(TempQuartFit-TempExact)

% let's see the quartic fit in a plot
TempQuartFitPlot=polyval(CoefQuartFit,timePlot);
plot(timePlot,TempExactPlot,'--k',...
     timeMeasured,TempMeasured,'ok',...
     timePlot,TempQuartFitPlot,'c')
legend('Exact','Measured','Quartic fit')
title('Least-Square Approximation with a Quartic')
xlabel('t (minutes)')
ylabel('T (Centigrade)')

% print the error
errQuartFitPlot=max(abs(TempQuartFitPlot-TempExactPlot))
disp(' ')
```

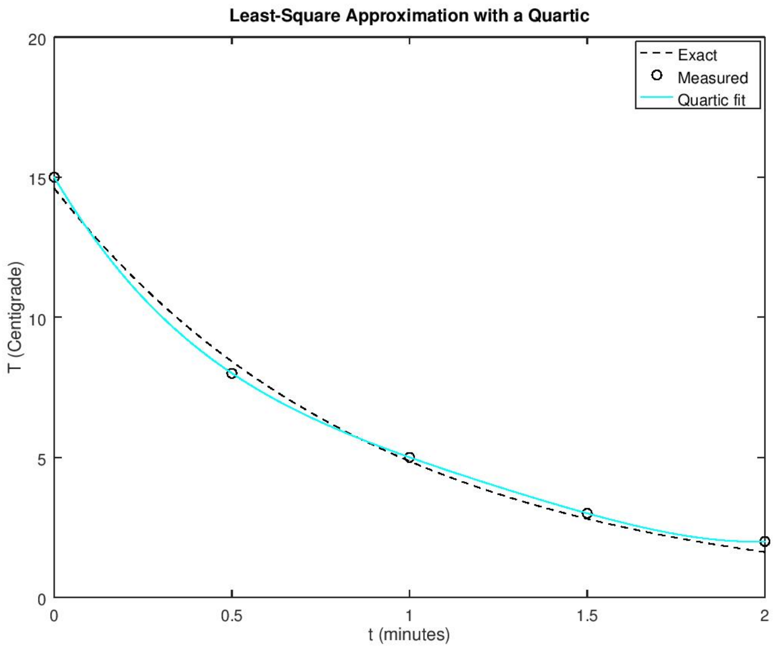
```
CoefQuartFit =
     2.0000   -10.0000    19.5000   -21.5000    15.0000
time =  0.70000
TempExact =  6.7600
TempQuartFit =  6.5552
errQuartFit =  0.20479
errQuartFitPlot =  0.47325
```

Extrapolation again

We already saw that extrapolation, i.e. evaluating outside the given range is fraught with peril. Let's try the fitted polynomials now.

```
% extrapolate again at t = 5
time=5
TempExact=TempExactFun(time)
TempLinear=interp1(timeMeasured,TempMeasured,time,...
                   'linear','extrap')
TempSpline=spline(timeMeasured,TempMeasured,time)
TempParFit=polyval(CoefParFit,time)
TempQuartFit=polyval(CoefQuartFit,time)
disp('Obviously, the extrapolated results are no good.')
disp(' ')
```

```
time =  5
```




```
TempExact =   0.059667
TempLinear = -4
TempSpline =   59
TempParFit =   39.371
TempQuartFit =   395.00
Obviously, the extrapolated results are no good.
```

SKIP: Fitting an exponential

According to the above, fitting a polynomial of at least quadratic degree worked reaonably well. But as noted earlier, it should be a much better idea to fit an exponential of the form

$$T = A\exp(Bt)$$

to our five data points. The reason is that the *exact* temperature is of the form above. You only need to get A (14.6) and B (-1.1) right, and you will get the right temperature, even in extrapolation.
The reason we did so far not try this is because the above expression is nonlinear in A and B. Then Matlab's `polyfit` function does not work.
However, we can apply a trick. If we take a natural logarithm of the expression above, we get:

$$\ln(T) = \ln(A) + Bt$$

Defining new variables as

$$C_1 = B \qquad C_2 = \ln(A)$$

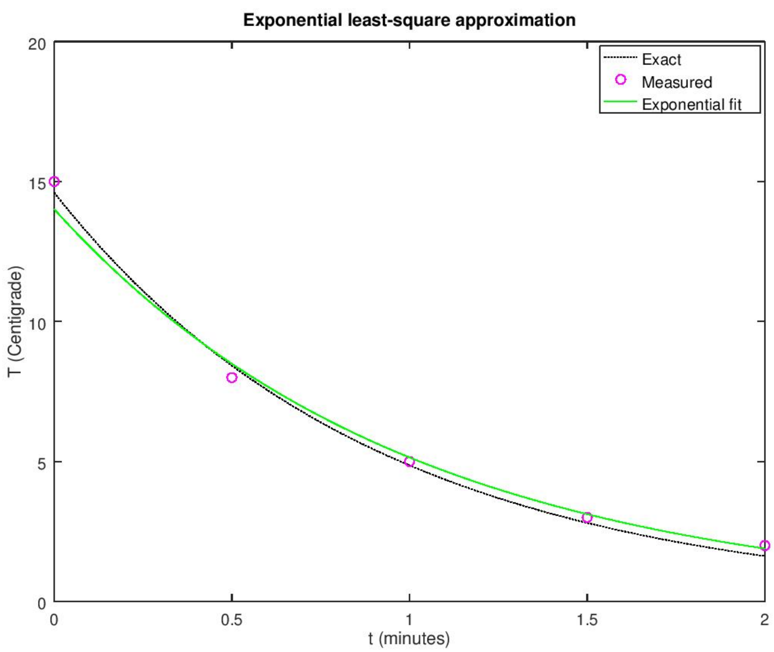
this takes the form

$$\ln(T) = C_1 t + C_2$$

That is just fitting by a straight line, but for $\ln(T)$ instead of T! The latter is not a problem; when we have T, we can find $\ln(T)$ by just taking a logarithm. And when we have $\ln(T)$, we can find T by just taking an exponential.
Below we try this out. Note that Matlab uses `log` for ln (and `log10` for log).

```
% create the measured ln(T) values
lnTempMeasured=log(TempMeasured);

% find C1 and C2
n=1;
CoefExpFit=polyfit(timeMeasured,lnTempMeasured,n)

% interpolate again at t = 0.7
time=0.7
% note the exp to convert ln(T) to T
TempExpFit=exp(polyval(CoefExpFit,time))
TempExact=TempExactFun(time)

% let's see the exponential fit in a plot
TempExpFitPlot=exp(polyval(CoefExpFit,timePlot));
plot(timePlot,TempExactPlot,':k',...
      timeMeasured,TempMeasured,'om',...
      timePlot,TempExpFitPlot,'g')
legend('Exact','Measured','Exponential fit')
title('Exponential least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
errExpFitPlot=max(abs(TempExpFitPlot-TempExactPlot))
disp('A bit disappointing, maybe.')
disp(' ')

% extrapolate again at t = 5
time=5
TempExact=TempExactFun(time)
TempExpFit=exp(polyval(CoefExpFit,time))
disp(' ')
```

```
CoefExpFit =
   -1.0021    2.6399
time =  0.70000
TempExpFit =  6.9475
TempExact =  6.7600
errExpFitPlot =  0.58869
A bit disappointing, maybe.

time =  5
TempExact =  0.059667
TempExpFit =  0.093409
```



SKIP: More on the exponential fit

The reason that the maximum error in the exponential fit is not much better than the quadratic one has to do with our manipulations. Since we changed unknowns to $\ln(T)$, Matlab is no longer making the average error in T as small as possible. It is now making the average error in $\ln(T)$ as small as possible.
This can be good or bad, depending on conditions. The error in T is the "absolute" error in the temperature. The error in $\ln(T)$ is the "relative" error in the temperature; the error relative to the magnitude of T. In other words,

the error in $\ln(T)$ gives the percentage error in T. Sometimes you would rather have the relative error as small as possible, instead of the absolute error.
On the other hand, if we are really interested in getting the smallest absolute error, there is a trick. Note first that the intial temperature is about 6 times bigger than the final temperature. To force Matlab to give more attention to that larger value, we can put it inside the measured data lists 6 times. Similarly, the value at t = 0.5 is about 3 times the final value and the one at t = 1 about 2 times, So we place these data that many times in the measured data lists.
Another way to achieve a similar effect would be to concentrate the measurements near the start, where the temperature is largest. But we will assume that the available measurements are as given.

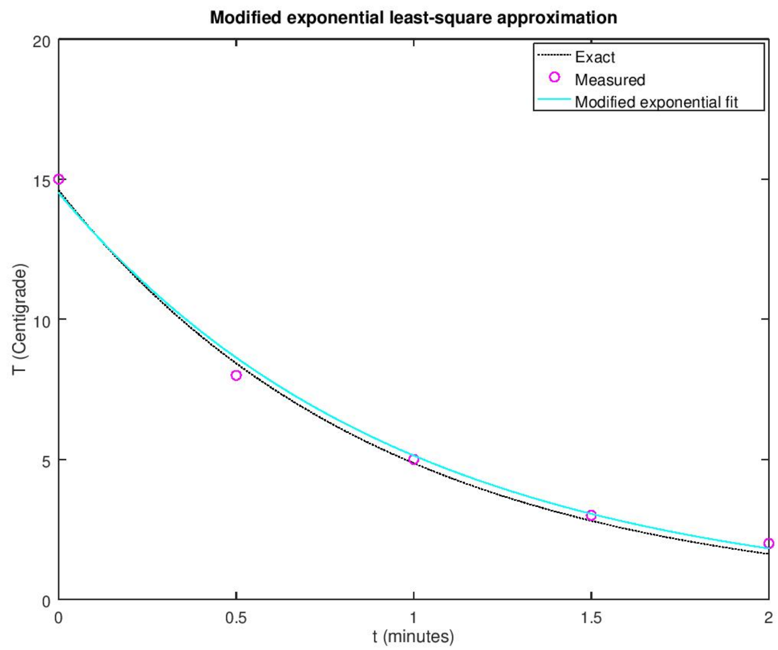
```
% let's check relative errors (note the ./)
RelErrParFit=max(...
    abs(TempParFitPlot-TempExactPlot)./TempExactPlot)
RelErrExpFit=max(...
    abs(TempExpFitPlot-TempExactPlot)./TempExactPlot)
disp('The relative error is much better!')

% lets improve the absolute error using the trick now
timeMeasuredMod = ...
    [ 0  0  0  0  0  0 0.5 0.5 0.5 1 1 1.5 2]';
TempMeasuredMod = ...
    [15 15 15 15 15 15  8   8   8  5 5  3  2]';
lnTempMeasuredMod=log(TempMeasuredMod);
n=1;
CoefExpFitMod=...
    polyfit(timeMeasuredMod,lnTempMeasuredMod,n)

% interpolate again at t = 0.7
time=0.7
TempExpFitMod=exp(polyval(CoefExpFitMod,time))
TempExact=TempExactFun(time)

% compare the interpolations in a plot
TempExpFitModPlot=exp(polyval(CoefExpFitMod,timePlot));
plot(timePlot,TempExactPlot,':k',...
     timeMeasured,TempMeasured,'om',...
     timePlot,TempExpFitModPlot,'c')
legend('Exact','Measured','Modified exponential fit')
title('Modified exponential least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
disp('Not too bad.')
disp(' ')
```

```
% compare the _maximum_ deviations
errExpFitModPlot=...
    max(abs(TempExpFitModPlot-TempExactPlot))
disp('The error is now much smaller than anything seen')
disp('before.  And we also follow the slope of the')
disp('exact curve very well.')
disp(' ')

% extrapolate again at t = 5
time=5
TempExact=TempExactFun(time)
TempExpFit=exp(polyval(CoefExpFit,time))
TempExpFitMod=exp(polyval(CoefExpFitMod,time))
disp('Not too bad.')
disp(' ')
```

```
RelErrParFit =  0.39526
RelErrExpFit =  0.16718
The relative error is much better!
CoefExpFitMod =
  -1.0388    2.6746
time =  0.70000
TempExpFitMod =  7.0102
TempExact =  6.7600
Not too bad.

errExpFitModPlot =  0.27342
The error is now much smaller than anything seen
before.  And we also follow the slope of the
exact curve very well.

time =  5
TempExact =  0.059667
TempExpFit =  0.093409
TempExpFitMod =  0.080485
Not too bad.
```

MORE MEASUREMENTS

If we would measure a lot more points than the 5 we have, and the errors in these measurements would be random, we could get a much better approximation. Unfortunately, rounding of temperatures to whole degrees is **not** random. It creates a deterministic "staircase" of numbers. But we can try anyway.
Note: We will cheat, and use the exact solution, which we are not supposed to know, to avoid doing and typing in 41 measurements.




```
% create the new 41 "measured" data
timeMeasured2=linspace(0,2,41);
% 'round' rounds to whole numbers
TempMeasured2=round(TempExactFun(timeMeasured2));

% use some more plot points now too
timePlot2=linspace(0,2,300);
TempExactPlot2=TempExactFun(timePlot2);
disp(' ')
```

Interpolation with more data

Note that the interpolations do *not* improve if we use more noisy points.

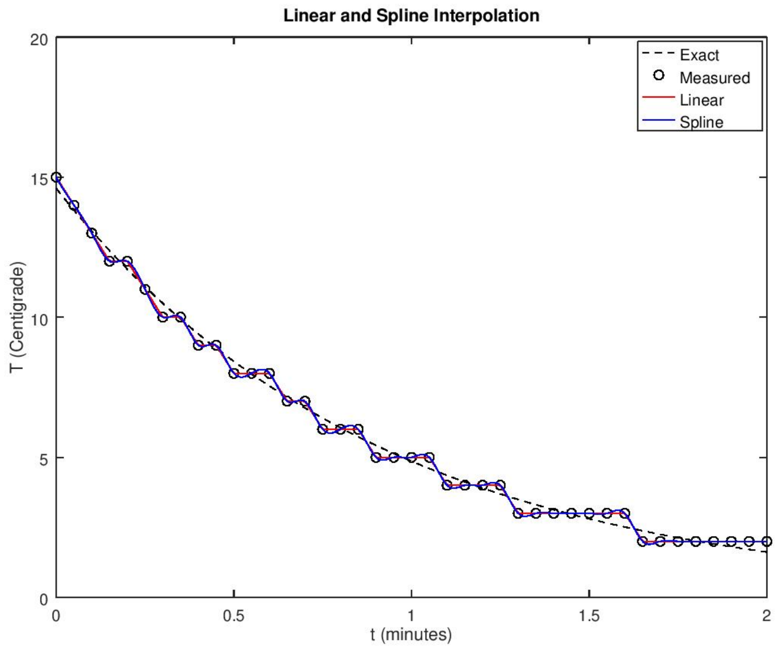
```
% compare the interpolations in a plot
TempLinearPlot2=...
    interp1(timeMeasured2,TempMeasured2,timePlot2);
TempSplinePlot2=...
    spline(timeMeasured2,TempMeasured2,timePlot2);
plot(timePlot2,TempExactPlot2,'--k',...
     timeMeasured2,TempMeasured2,'ok',...
```

```
    timePlot2,TempLinearPlot2,'r',...
    timePlot2,TempSplinePlot2,'b')
legend('Exact','Measured','Linear','Spline')
title('Linear and Spline Interpolation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
disp('Seems worse than before.')
disp(' ')

% compare the _maximum_ deviations
errLinearPlot
errLinearPlot2=max(abs(TempLinearPlot2-TempExactPlot2))
errSplinePlot
errSplinePlot2=max(abs(TempSplinePlot2-TempExactPlot2))
disp('Now the spline is worse that linear!')
disp(' ')
```

```
Seems worse than before.

errLinearPlot =  0.52550
errLinearPlot2 =  0.48471
errSplinePlot =  0.44453
errSplinePlot2 =  0.55055
Now the spline is worse that linear!
```

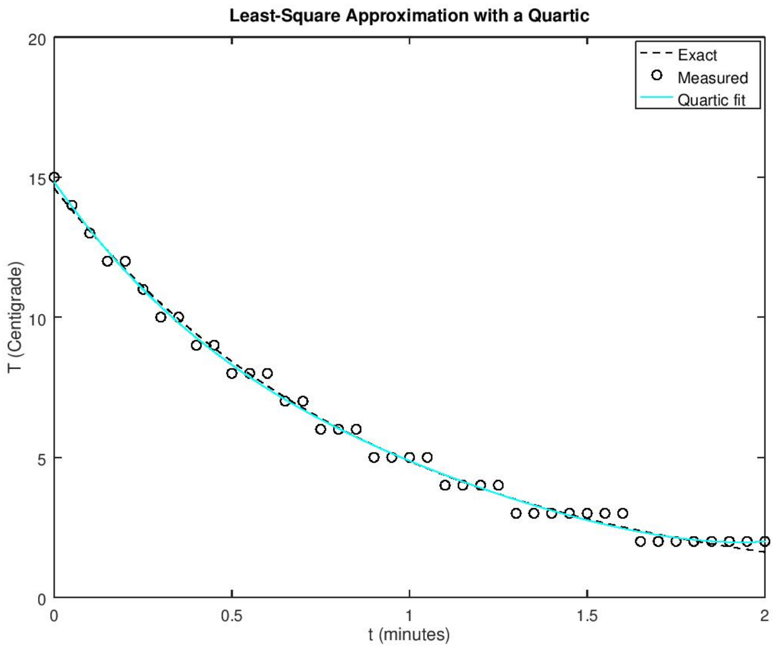
Quartic fit with more data

```
% repeat the quartic fit
n=4;
CoefQuartFit2=polyfit(timeMeasured2,TempMeasured2,n)

% compare the interpolations in a plot
TempQuartFitPlot2=polyval(CoefQuartFit2,timePlot2);
plot(timePlot2,TempExactPlot2,'--k',...
    timeMeasured2,TempMeasured2,'ok',...
    timePlot2,TempQuartFitPlot2,'c')
legend('Exact','Measured','Quartic fit')
title('Least-Square Approximation with a Quartic')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
disp('Over most of the range, this is much better than')
disp('the result for 5 measured values (which went')
disp('through all measured points).')
disp(' ')
```




```
% compare the _maximum_ deviations
errQuartFitPlot
errQuartFitPlot2=...
    max(abs(TempQuartFitPlot2-TempExactPlot2))
disp('The disappointing maximum error is due to the')
disp('fact that the final "measured" points are all too')
disp('high.  Have a good look at the end of the graph!')
disp(' ')
```

```
CoefQuartFit2 =
    1.2249   -6.0707   13.1791   -18.2949   14.8320
Over most of the range, this is much better than
the result for 5 measured values (which went
through all measured points).

errQuartFitPlot =  0.47325
errQuartFitPlot2 =  0.37455
The disappointing maximum error is due to the
fact that the final "measured" points are all too
high.  Have a good look at the end of the graph!
```

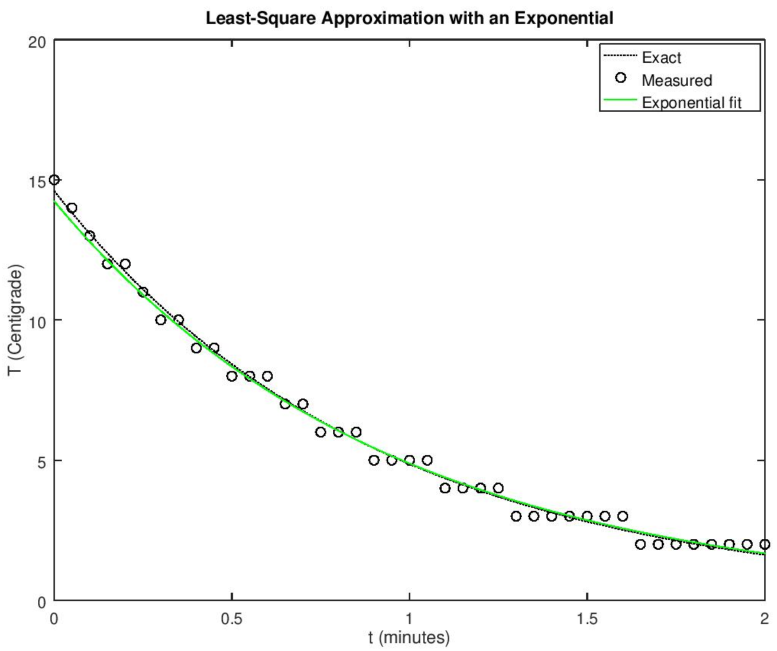


SKIP: Exponential fit with more data

```
% repeat the exponential fit
lnTempMeasured2=log(TempMeasured2);
n=1;
CoefExpFit2=polyfit(timeMeasured2,lnTempMeasured2,n)

% compare the interpolations in a plot
TempExpFitPlot2=exp(polyval(CoefExpFit2,timePlot2));
plot(timePlot2,TempExactPlot2,':k',...
     timeMeasured2,TempMeasured2,'ok',...
     timePlot2,TempExpFitPlot2,'g')
legend('Exact','Measured','Exponential fit')
title('Least-Square Approximation with an Exponential')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
disp(' ')

% compare the _maximum_ deviations
errExpFitPlot
errExpFitPlot2=max(abs(TempExpFitPlot2-TempExactPlot2))
disp('Clearly, that is quite good.')
```

```
disp(' ')
```

```
CoefExpFit2 =
  -1.0704    2.6558

errExpFitPlot =  0.58869
errExpFitPlot2 =  0.36353
Clearly, that is quite good.
```



INTEGRALS

It is easy to do determined integrals, with given limits, using Matlab. Just use the `integral` function. (Octave still uses the old name `quad`.)
As an *example* (which you do not actually have to understand), suppose we want to know how much radiation q the bar in our example emits per unit surface area while cooling down. Assuming that the bar surface is perfectly black, the Stefan-Boltzmann law says that the radiation emitted per unit area and unit time is given by

$$\dot{q} = \sigma T^4 \qquad \sigma = 5.670373\,10^{-8}\ \mathrm{W/m^2K^4}$$

where T is the *absolute* temperature in K (Kelvin) and σ is called the "Stefan-Boltzmann constant." To get the q we want, we need to integrate $\dot{q}$ above from time 0 to time 2:

$$q = \int_0^2 \sigma T^4 \, \mathrm{d}t$$

If you do not understand why, that is OK. All you need to know is that

1. We want to do the integral above.
2. Since T above is in K, we must add $T_0 = 273.15$ to our temperatures, which are in Centigrade. Also we need to insert a factor 60 to convert our times from minutes to seconds.
3. In Matlab integration is nowadays done by a function called `integral`. Octave still uses the old name `quad`.
4. Function `integral` (or `quad`) can only integrate *functions*. You *cannot* use it to integrate measured data or plot points! (There are different methods to do that, like "spline integration", but we do not cover those here.)

```
% the Stefan-Boltzmann constant in W/m^2 K^4:
sigma=5.670373E-8;

% 0 degrees Centigrade in Kelvin
T0=273.15;
```

The exact integral

Let's find the exact integral first using our knowledge of Calculus I. To integrate

$$q = \int_0^2 \sigma \left(A \exp(Bt) + T_0 \right)^4 \, \mathrm{d}t \qquad A = 14.6, B = -1.1$$

with respect to time, change integration variable to $u = A \exp(Bt)$ and take it from there.

```
% names for the constants in TempExact = A exp(B t)
A=14.6;
B=-1.1;

% evaluate the start (t=0) and end (t=2) values of u
u1=A;
u2=A*exp(B*2);

% evaluate the integral as found by calculus
```

```
qTrue=60*sigma*(...
    1/4*(u2^4-u1^4)+...
    4/3*T0*(u2^3-u1^3)+...
    3*T0^2*(u2^2-u1^2)+...
    4*T0^3*(u2-u1)+...
    T0^4*(log(u2)-log(u1)))/B;
fprintf('Truly exact: %.3f\n',qTrue)
```

```
Truly exact: 41301.515
```

Numerical integrations

Next let's use numerical integration, i.e. `integral` or `quad`, to find the integral. Note that there will be an error created by the numerical integration, even if we integrate the exact temperature.
We already noted before that `integral` or `quad` can only integrate a *function*. So typically, we will provide it an anonymous function of the time t, in

```
@(t) FUNCTION(..., t, ...)
```

notation, where `t` is the time and dots stand for other parameters of `FUNCTION`. Matlab functions that can be used to create `FUNCTION` are `TempExactFun`, `interp1`, `spline`, and `polyval` (after their other parameters have been found). Do not try to put in arrays (like ...`Measured` or ...`Plot`) except as parameters of these functions!
Note that **all** obtained values below will be pretty accurate:

```
Typically:
   Numerical errors tend to become less in integrals.
```

```
% try numerical integration of the exact temperature
qExact=60*sigma*quad(...
   @(t) (TempExactFun(t)+T0).^4,0,2);
fprintf('Numerical integration: %.3f Error: %.1E%%\n',...
        qExact,abs(qExact-qTrue)/qTrue*100)
disp('As shown, numerical integration for a smooth')
disp('function like this will be very accurate.')
disp('The error is smaller than the round-off.')

% try numerical integration of the linear interpolation
qLinear=60*sigma*quad(...
  @(t) (interp1(timeMeasured,TempMeasured,t)+T0).^4,0,2);
fprintf('Linear interpolation:  %.3f Error: %.3f%%\n',...
        qLinear,abs(qLinear-qTrue)/qTrue*100)

% try numerical integration of the spline interpolation
```

```
qSpline=60*sigma*quad(...
   @(t) (spline(timeMeasured,TempMeasured,t)+T0).^4,0,2);
fprintf('Spline interpolation:  %.3f Error: %.3f%%\n',...
         qSpline,abs(qSpline-qTrue)/qTrue*100)

% try numerical integration of the parabolic fit
qParFit=60*sigma*quad(...
   @(t) (polyval(CoefParFit,t)+T0).^4,0,2);
fprintf('Parabolic Fit:         %.3f Error: %.3f%%\n',...
         qParFit,abs(qParFit-qTrue)/qTrue*100)

% try numerical integration of the quartic fit
qQuartFit=60*sigma*quad(...
   @(t) (polyval(CoefQuartFit,t)+T0).^4,0,2);
fprintf('Quartic Fit:           %.3f Error: %.3f%%\n',...
         qQuartFit,abs(qQuartFit-qTrue)/qTrue*100)
disp(' ')
```

```
Numerical integration: 41301.515 Error: 0.0E+00%
As shown, numerical integration for a smooth
function like this will be very accurate.
The error is smaller than the round-off.
Linear interpolation:  41434.104 Error: 0.321%
Spline interpolation:  41307.239 Error: 0.014%
Parabolic Fit:         41352.074 Error: 0.122%
Quartic Fit:           41297.198 Error: 0.010%
```

SKIP: More on integration

In this section we will show how the exponential curve fits work out.
We will also show how you can integrate the polynomial fits *exactly* if you want. Not only does this eliminate all integration errors, however small, Matlab can also do it much more quickly. That could be important if you have to do a lot of these integrals. The trick is to use function `polyint` instead of `integral` or `quad` to do the integration.

```
% try numerical integration of the exponential fit
B=CoefExpFit(1);
lnA=CoefExpFit(2);
qExpFit=quad(@(t) (exp(B*t+lnA)+T0).^4,0,2)*60*sigma;
fprintf('Exponential Fit:       %.3f Error: %.3f%%\n',...
         qExpFit,abs(qExpFit-qTrue)/qTrue*100)

% try numerical integration of the tricked one too
B=CoefExpFitMod(1);
lnA=CoefExpFitMod(2);
```

```
qExpFitMod=quad(@(t) (exp(B*t+lnA)+T0).^4,0,2)*60*sigma;
fprintf('Exponential Fit, Mod:  %.3f Error: %.3f%%\n',...
        qExpFitMod,abs(qExpFitMod-qTrue)/qTrue*100)

% integrate the parabolic fit exactly now
disp('Exact integration of the parabolic fit:')
% first convert the polynomial to Kelvin by adding T0
tempC=CoefParFit;
tempC(end)=tempC(end)+T0;
% now square that polynomial twice to get T^4
tempC=conv(tempC,tempC);
tempC=conv(tempC,tempC);
% find the antiderivative polynomial using polyint
tempC=polyint(tempC);
% evaluate the integral between 0 and 2 with that
tempInt=polyval(tempC,2)-polyval(tempC,0);
% add the remaining factors
qParExact=tempInt*60*sigma;
fprintf('Quadratic Fit, exact:  %.3f Error: %.3f%%\n',...
        qParExact,abs(qParExact-qTrue)/qTrue*100)

% integrate the quartic fit the same way
disp('Exact integration of the quartic fit:')
% first convert the polynomial to Kelvin by adding T0
tempC=CoefQuartFit;
tempC(end)=tempC(end)+T0;
% now square that polynomial twice to get T^4
tempC=conv(tempC,tempC);
tempC=conv(tempC,tempC);
% find the antiderivative polynomial using polyint
tempC=polyint(tempC);
% evaluate the integral between 0 and 2 with that
tempInt=polyval(tempC,2)-polyval(tempC,0);
% add the remaining factors
qQuartExact=tempInt*60*sigma;
fprintf('Quartic Fit, exact:    %.3f Error: %.3f%%\n',...
        qQuartExact,abs(qQuartExact-qTrue)/qTrue*100)
disp(' ')
```

```
Exponential Fit:        41384.031 Error: 0.200%
Exponential Fit, Mod:   41422.232 Error: 0.292%
Exact integration of the parabolic fit:
Quadratic Fit, exact:   41352.074 Error: 0.122%
Exact integration of the quartic fit:
Quartic Fit, exact:     41297.198 Error: 0.010%
```

DERIVATIVES

Sometimes we are interested in the derivative of the quantity in question. In the present example, it is a measure of how much heat leaks out of the bar per unit time.

The exact derivative

Since

$$T_{\text{exact}} = A\exp(Bt) \qquad A = 14.6, B = -1.1$$

its derivative is simply

$$\frac{\mathrm{d}T_{\text{exact}}}{\mathrm{d}t} = BT_{\text{exact}} \qquad B = -1.1$$

(That follows from differentiating the exponential using the chain rule.)

```
% derivative of TempExact found analytically
derTempExactPlot=-1.1*TempExactPlot;
```

Numerical differentiation

For the linear, quadratic, and quartic fits, we can use the fact that Matlab function `polyder` will find the coefficients of the derivative polynomial for us. Then we can use our old faithful `polyval` to evaluate that derivative polynomial. In this section we will use these methods to find the derivative of the temperature at our plot points, and then plot the results. We will compare the quadratic for 5 noisy measurements and the quartic for 41.
Note that the results will be pretty bad.

```
Typically:
   Errors tend to become much worse in derivatives.
```

We will also plot the quartic for 41 noisy data points when the noise is random, rather than due to a systematic rounding error. This will allow the method of least squares to work like it is designed for. The results will be much better.

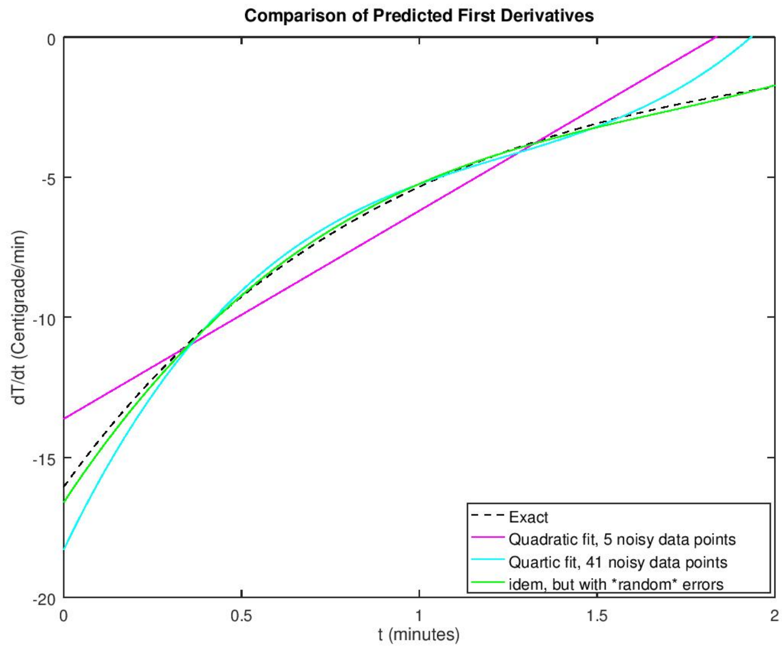
```
% derivative of the parabolic fit polynomial
derCoefParFit=polyder(CoefParFit);
% use it to evaluate the derivative at the plot points
derTempParFitPlot=polyval(derCoefParFit,timePlot);

% derivative of the quartic fit polynomial, 41 points
derCoefQuartFit2=polyder(CoefQuartFit2);
% use it to evaluate the derivative at the plot points
derTempQuartFitPlot2=polyval(derCoefQuartFit2,timePlot2);
```

```
% quartic fit, 41 points with random errors
randn("seed",9);
TempMeasured3=...
    TempExactFun(timeMeasured2)+0.25*randn(1,41);
n=4;
CoefQuartFit3=polyfit(timeMeasured2,TempMeasured3,n)
% derivative of the quartic fit polynomial, 41 points
derCoefQuartFit3=polyder(CoefQuartFit3);
% use it to evaluate the derivative at the plot points
derTempQuartFitPlot3=polyval(derCoefQuartFit3,timePlot2);

% plot it
plot(timePlot,derTempExactPlot,'--k',...
     timePlot,derTempParFitPlot,'m',...
     timePlot2,derTempQuartFitPlot2,'c',...
     timePlot2,derTempQuartFitPlot3,'g')
axis([0 2 -20 0])
legend('Exact',...
       'Quadratic fit, 5 noisy data points',...
       'Quartic fit, 41 noisy data points',...
       'idem, but with *random* errors')
legend('location','southeast')
title('Comparison of Predicted First Derivatives')
xlabel('t (minutes)')
ylabel('dT/dt (Centigrade/min)')
disp('The polynomial fit derivatives are pretty bad.')
disp('The quartic is no better than the quadratic.')
disp('However, if the noise is really random, the')
disp('quartic can become quite good for a lot of')
disp('data points.')
disp(' ')
```

```
CoefQuartFit3 =
    0.48051   -3.22981    9.56650   -16.60748   14.70296
The polynomial fit derivatives are pretty bad.
The quartic is no better than the quadratic.
However, if the noise is really random, the
quartic can become quite good for a lot of
data points.
```

SKIP: More on differentiation

In this section we show the derivatives of the exponential fits obtained earlier. As expected, these are much more reasonable. Both the exponential fit for 41



noisy measurements, as well as the modified fit for 5 noisy measurements are plotted.

How about the derivative of your beloved interpolated spline? Well, linear and spline interpolation are described by "piecewise polynomials": there is a different polynomial in each segment between measured points. The bad thing is that the idiots at MathWorks never defined a function to find the derivatives of piecewise polynomials. If you want the derivative of your spline, look for 'ppder' or 'ppdiff' provided by third parties, (where pp is an acronym for "piecewise polynomial".) Octave provides ppder. This is used below to plot the derivative of the spline with 5 exact measurements (pretty good). It is also used to plot the derivative of the spline with 41 noisy measurements (very bad).

```
Remember:
  Spline differentiation might be good,
    but noisy data are a big problem.
```

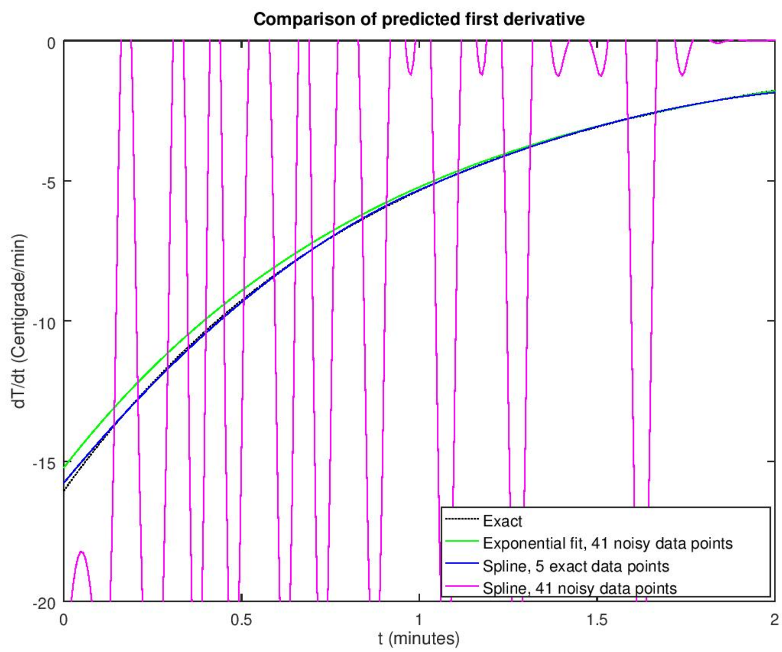
Note added 9/22/2018: Matlab R2018b now seems to have `fnder` for derivatives (and `fnint` for integrals) of piecewise polynomials. Unfortunately, we are still using Matlab R2017b at the time of writing.

```
% derivatives of TempExpFit2
derTempExpFitPlot2=CoefExpFit2(1)*TempExpFitPlot2;

% piecewise polynomial coefficients of the spline
```

```
ppSpline=spline(timeMeasured,TempExactFun(timeMeasured));
% find the coeeficients of the derivative
derppSpline=ppder(ppSpline);
% evaluate at the plot points
derTempSplinePlot=ppval(derppSpline,timePlot);

% piecewise polynomial coefficients of the spline
ppSpline2=spline(timeMeasured2,TempMeasured2);
% find the coeeficients of the derivative
derppSpline2=ppder(ppSpline2);
% evaluate at the plot points
derTempSplinePlot2=ppval(derppSpline2,timePlot2);

% plot it
plot(timePlot,derTempExactPlot,':k',...
     timePlot2,derTempExpFitPlot2,'g',...
     timePlot,derTempSplinePlot,'b',...
     timePlot2,derTempSplinePlot2,'m')
axis([0 2 -20 0])
legend('Exact',...
       'Exponential fit, 41 noisy data points',...
       'Spline, 5 exact data points',...
       'Spline, 41 noisy data points')
legend('location','southeast')
title('Comparison of predicted first derivative')
xlabel('t (minutes)')
ylabel('dT/dt (Centigrade/min)')
disp('The exponential fit derivative is reasonable.')
disp('The spline is good for exact data, but noisy')
disp('data can be a big problem.')
disp(' ')
```

```
The exponential fit derivative is reasonable.
The spline is good for exact data, but noisy
data can be a big problem.
```

End lesson 3


Comparison of predicted first derivative
0
-5
-10
-15
-20
0
0.5
1
1.5
2
t (minutes)
dT/dt (Centigrade/min)
Exact
Exponential fit, 41 noisy data points
Spline, 5 exact data points
Spline, 41 noisy data points