3 INTERPOLATION

Contents

```
% make sure the workspace is clear
if ~exist('____code____','var') ; clear ; end
% reduce needless whitespace
format compact
% reduce irritations (pausing and buffering)
more off
% start a diary (in the actual lecture)
%diary lectureN.txt

% Tell the students to save their work space before
% leaving: save lectureN
```

LESSON SUMMARY

- The big question in this lesson is the following: Given a set of measured values (the data) how do you find the values of an unknown you want at conditions that are not quite the same as any that were measured? Or equivalently, given a table of values (the data), how do you find a value you want at conditions that are not quite the same as the ones given in the table?

- The most straightforward answer is "interpolation". You probably already know how to do "linear interpolation" with a calculator yourself. Matlab can do the same thing for you using the `interp1` function. Linear interpolation requires at least two data points around the desired value.

- Matlab can also do "spline interpolation" using the `spline` function. Assuming the data points are accurate, reasonably closely spaced, and there are enough of them, spline interpolation can be *much* more accurate than linear interpolation.

- If the available data do have significant errors (as often happens for experimental data), linear and spline interpolation may not work so well. In that case it may be a better idea to approximate ("fit") the data by a curve of a given shape, say a straight line, or a quadratic, or whatever. Normally, the selected curve does not pass exactly through all the data points. But that is OK, as these data points have errors. If there are enough data points, and the errors in them are random, the fitted curve may be a lot more accurate than linear or spline interpolation.

- The simplest is to fit a polynomial (a straight line, or a quadratic, or a cubic, or a quartic, or a quintic, etcetera) to the data. Matlab function `polyfit` can do that for you. You simply give `polyfit` the data points and the degree of the desired polynomial.
- Function `polyfit` gives you the *coefficients* of the fitted polynomial. To evaluate the polynomial at the desired position(s), you must use a second function, `polyval`. You simply give `polyval` the coefficients of the polynomial and the desired position(s).
- Often, it may be a better idea to fit a curve different from a polynomial to the data. In the example used in this lesson, fitting an exponential would be a better choice. However, we will skip the parts on how to fit an exponential. If you need it in future, you will have to read that section of this lesson on your own. Later lesson 8 will cover how to fit a power relationship to data.
- Sometimes you may need to integrate or differentiate the measured or tabulated quantity. To integrate numerically, use the `quad` function. Or in recent versions of Matlab (but not Octave) you can use the `integral` function. Both `quad` and `integral` need a function to integrate, not measured or tabulated data. You can create the needed function by pushing `spline` or `polyval` into a suitable anonymous function.
- While integration is usually not a big deal, differentiation is a very different story. It is typically hard to do it well. If you use a polynomial fit, `polyder` will give you the coefficients of the derivative polynomial. You can then evaluate that using `polyval`.
- In principle, you can also differentiate the interpolated spline. However, we will skip that. If you need it in future, you will have to read that section of this lesson on your own.

Key areas of the online book

Before the first lecture, in the online book do:

- 4.2 Row arrays: complete the section.
- 4.3 Constructing row arrays: complete the section.
- 18.1 Interpolation: skip the stuff beyond CA 18.1.1.

Also make sure you have read the "Read: ..." section on function handles of lesson 2.

Before the second lecture, in the online book do:

- 4.4 Multi-element row array indexing ...: the PAs.
- 18.2 Curve fitting - Least squares regression: skip the stuff beyond CA 18.2.1. In CA 18.2.1, note that only data points for speeds between 6 and 12 should be used, so you need a subset of the available data points. See section 4.4. If the math of this poorly written section leaves you clueless, see whether a peek at the lesson below helps.

While the online book has very mathematical sections 18.3 on integration and 18.4 on differentiation, I doubt they will be helpful to you. Too much math that is not useful for real-life applications.

INTERPOLATION.

Probably, you have already done interpolation before in other courses. The next few sections will explain how you can do it much easier and better with Matlab.

The example problem

As an example, initially we will use the following table of "measured" data to interpolate:

```
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 above five measured times and temperatures respectively.

OK, these data are not really measured; I made them up. But they are *like* data you might actually encounter in engineering problems. For example, if you have a hot solid, like a hot metal bar, and you let it cool back down to the ambient temperature, then in the later stages of the cool down process, the temperature in the middle of the bar, relative to ambient, may be given by a relation like the one above. In any case, all you need to know to do interpolation is the numbers to interpolate. Whatever these numbers mean physically is irrelevant.

```
% 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]
```

```
timeMeasured =
   0.00000    0.50000    1.00000    1.50000    2.00000
TempMeasured =
   14.6000     8.4200     4.8600     2.8000     1.6200
```

What we do not know (or pretend to)

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 can create a function `TempExactFun` to evaluate the exact temperature that we pretend not to know. But since the function is very simple and not intended for more general use, it is not worth it to create a function file for it. Instead we can define `TempExactFun` as a "handle" to an anonymous function. See lesson 2 for more.

```
% make TempExactFun a handle to an anonymous function
TempExactFun = @(t) 14.6*exp(-1.1*t)
```

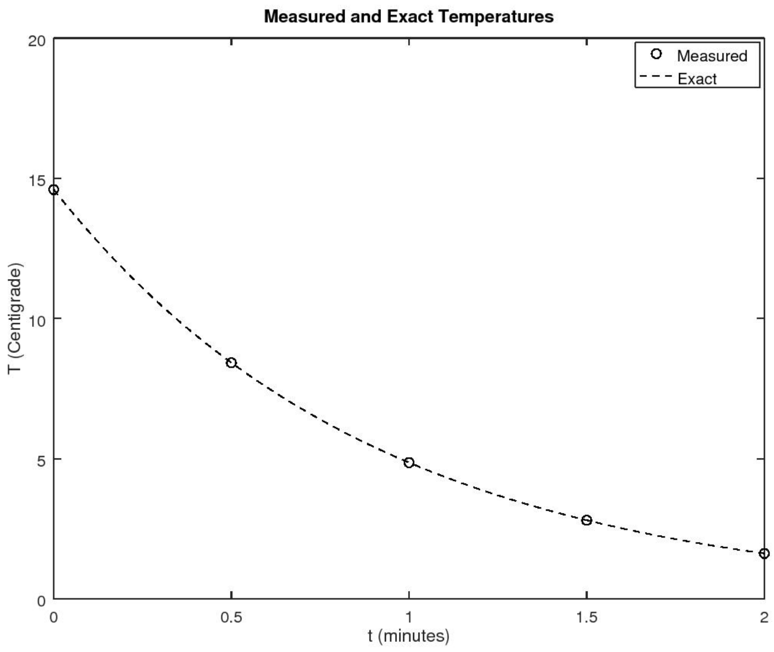
```
TempExactFun =
@(t) 14.6 * exp (-1.1 * t)
```

Plot to understand the problem better

Let's plot the measured five values as data points in a graph. In the same graph, let's also plot the exact solution that we pretend not to know, as a curve. Remember from lesson 2 that to plot a curve, you need to create a set of plot points. These are *different* from the measured points and they are only used for plotting.

```
% generate 301 plot time values between 0 and 2
timePlot=linspace(0,2,301);
% generate corresponding exact temperatures
TempExactPlot=TempExactFun(timePlot);

% create the plot, using black circles for the measured
% points and a black broken line for the exact solution
% that supposedly we do not know
plot(timeMeasured,TempMeasured,'ok',...
     timePlot,TempExactPlot,'--k')
legend('Measured','Exact')
title('Measured and Exact Temperatures')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
```



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. Using interpolation can give us a value for the temperature at time 0.7. Ideally speaking, this value would be the same as the exact temperature that we pretend we do not know.

Matlab provides `interp1` or `spline` to interpolate.

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

```
timeDesired =  0.70000
TempLinear =  6.9960
TempSpline =  6.7513
```

```
% two reasonable values, but which one is best???
TempExact=TempExactFun(timeDesired)
```

```
errLinear=abs(TempLinear-TempExact)
errSpline=abs(TempSpline-TempExact)
```

```
TempExact =   6.7600
errLinear =   0.23601
errSpline =   0.0086708
```

For a nice smooth curve, spline interpolation is much more accurate than linear interpolation!

Plot the interpolated values for all times

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

The spline is right on top of the exact curve in the plot.

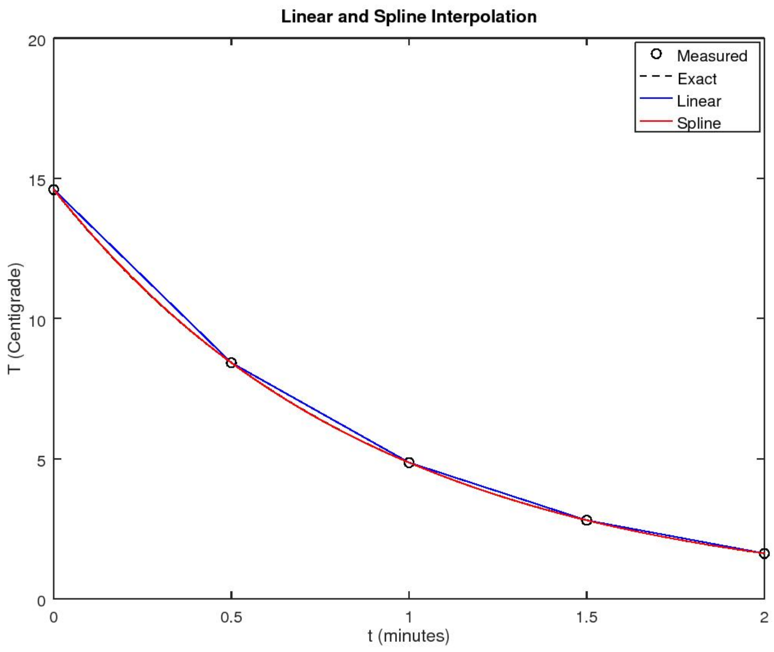
```
% compare the maximum errors
errLinearMax=max(abs(TempLinearPlot-TempExactPlot))
errSplineMax=max(abs(TempSplinePlot-TempExactPlot))
```

```
errLinearMax =   0.42119
errSplineMax =   0.017674
```

The spline is much more accurate.

Extrapolation

Suppose that the time at which we want to know the temperature is $t = 3$. This time is not inside the measured range from 0 to 2. If that happens, we talk about *extrapolation* instead of *interpolation*.



WARNING: *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 = 3
timeDesired=3
TempLinear=interp1(timeMeasured,TempMeasured,...
                   timeDesired,'linear','extrap')
TempSpline=spline(timeMeasured,TempMeasured,timeDesired)
% compare with exact
TempExact=TempExactFun(timeDesired)
```

```
timeDesired =  3
TempLinear = -0.74000
TempSpline = -0.080000
TempExact =  0.53849
```

Extrapolation is usually bad news!

Both the linear and spline extrapolated values are useless.

End conditions for 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 to 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` (see also the final section on spline differentiation).

NOISY DATA

What if the measured data have errors? Measurements are not exact, especially temperature ones. After correcting for systematic effects, you are likely to have random errors, "noise", of either sign left. To get an idea how big these errors are, you typically do more measurements. The true values are then probably somewhere in the middle of the measured values.

To study dealing with noisy data, we will now assume that there are 40, rather than 5 measurements available. However, we will also assume that these data have random errors of a typical magnitude of 0.5 degrees Centigrade.

More precisely, we will assume that the "root mean square" (RMS) error is 0.5 degrees. The RMS error is what you get if you first compute the average *square* error and then take the square root of that.

To avoid actually having to do 40 temperature measurements, we will cheat and just take the 40 temperatures from the exact solution we are not supposed to know. Then we add to that "random" errors produced with help from the Matlab function `randn`.

```
% average ("Root Mean Square") random error
errMeasuredRMS=0.5

% create the 40 "measured" times; N stands for "noisy"
timeMeasuredN=linspace(0,2,40);

% initialize the random number generators
%rng('default') % for Matlab
randn('seed',4); % for Octave; use 4 or 9, avoid 0, 2, 6

% create the "measured" data with random errors
TempMeasuredN=TempExactFun(timeMeasuredN)...
    +errMeasuredRMS*randn(size(timeMeasuredN))
```

```
errMeasuredRMS =  0.50000
TempMeasuredN =
 Columns 1 through 5:
   14.7433   13.7125   13.3868   12.5351   11.9426
 Columns 6 through 10:
   10.6263     9.9010     9.7527     8.7326     9.5423
 Columns 11 through 15:
    8.6245     7.7905     8.2066     7.0018     6.0363
 Columns 16 through 20:
    4.8542     6.3331     5.4712     6.0007     4.6928
 Columns 21 through 25:
    4.4802     4.5192     4.5513     3.7844     3.2279
 Columns 26 through 30:
    3.1180     3.5670     4.6049     2.6473     4.3204
 Columns 31 through 35:
    1.7274     1.8586     2.6555     2.1452     1.9262
 Columns 36 through 40:
    1.4569     2.3194     1.4742     1.0307     2.2034
```

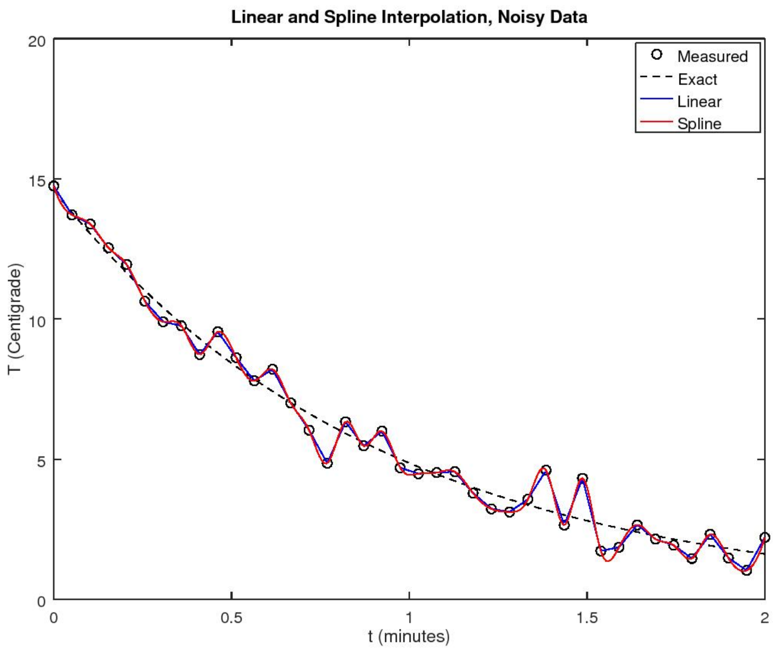
```
% interpolate again at t = 0.7
timeDesired=0.7
TempLinearN=...
    interp1(timeMeasuredN,TempMeasuredN,timeDesired)
TempSplineN=...
    spline(timeMeasuredN,TempMeasuredN,timeDesired)
TempExact=TempExactFun(timeDesired)
errLinearN=abs(TempLinearN-TempExact)
errSplineN=abs(TempSplineN-TempExact)
```

```
timeDesired =  0.70000
TempLinearN =  6.3742
TempSplineN =  6.4255
TempExact =  6.7600
errLinearN =  0.38581
errSplineN =  0.33448
```

There is no longer a real difference in error.

```
% compare the interpolations again in a plot
TempLinearNPlot=...
    interp1(timeMeasuredN,TempMeasuredN,timePlot);
TempSplineNPlot=...
    spline(timeMeasuredN,TempMeasuredN,timePlot);
plot(timeMeasuredN,TempMeasuredN,'ok',...
     timePlot,TempExactPlot,'--k',...
     timePlot,TempLinearNPlot,'b',...
     timePlot,TempSplineNPlot,'r')
legend('Measured','Exact','Linear','Spline')
title('Linear and Spline Interpolation, Noisy Data')
xlabel('t (minutes)')
```

```
ylabel('T (Centigrade)')
```



The spline looks crazier in the plot than linear interpolation.

```
% compare the maximum errors
errLinearNMax=max(abs(TempLinearNPlot-TempExactPlot))
errSplineNMax=max(abs(TempSplineNPlot-TempExactPlot))
```

```
errLinearNMax =   1.4583
errSplineNMax =   1.4764
```

The spline is now worse than linear.

```
% try extrapolation again
timeDesired=3
TempLinearN=interp1(timeMeasuredN,TempMeasuredN,...
                        timeDesired,'linear','extrap')
TempSplineN=...
     spline(timeMeasuredN,TempMeasuredN,timeDesired)
TempExact=TempExactFun(timeDesired)
```

```
timeDesired =   3
TempLinearN =   25.072
TempSplineN =   1066.2
TempExact =   0.53849
```

That is horrible!! Especially the spline!

SAVING AND RELOADING

(Note: all commands in this section have been disabled as they would interfere with publishing this lesson.)
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.

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

% test if it worked 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
```

Some things you may want to remember for future use:

1. To save only a few variables, you could use the `save FILENAME VAR1 VAR2 ...` command.

1. 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.

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 the errors.

If we want something more accurate than `interp1` and `spline`, we must drop the assumption that our interpolation reproduces all the inaccurate data *exactly*. The more accurate interpolation we want should be *close* to the measured data, but it should not reproduce all their errors. So it cannot go *exactly* through each measured point.

What we do in "curve fitting" is first choose a relatively simple function type (the curve) to represent the data. For example, for the data we are using in this lesson, the *exact* temperature curve is given by:

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

To be sure, we are assuming that we do not know that. But without knowing the exact solution, I would still be able to guess that the desired temperature is of the form

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

where A and B are unknown constants. And so should you, after you have finished your Heat Transfer class. For now, you will just have to take it from me. In any case, you must agree that this curve cannot swing wildly back and forward from point to point like linear and spline interpolation do.

To be sure, there is no way to guess the values of the constants A and B in the expression. But we can choose the values of A and B that give the best approximation to the data we do know. In other words, we can "fit" our exponential curve to the given data by selecting the constants A and B appropriately. This is sure to produce better results than `interp1` and `spline`.

Finding the constants: "least squares"

Of course, the devil is in the details. In particular, how exactly are you going to find the constants A and B that make the exponential curve fit the data best?

What does "fit the data best" mean in the first place? Obviously you want the differences between the fitted curve and the available data to be as small as possible, in some sense. One possible way to do that is to select the curve for which the *maximum absolute difference* among all the measured data is as small as possible. However, the usual approach is to instead make the *average square* difference as small as possible. This is called the "Method of Least Squares". There are two reasons why this method is popular:

1. Theoretically, in simple cases where the errors are truly random with everywhere the same typical magnitude, it gives the best possible answer.
2. Practically, the mathematics of making the average square difference as small as possible is a lot simpler than other possibilities like making the maximum absolute difference as small as possible.

We do not really need to worry about the latter anyway, as Matlab does that work for us.

But even with Matlab helping us, finding the best constants A and B in the exponential curve described so far is a bit tricky. So we will try to fit some simpler curves first.

Fitting a line

To keep things as simple as possible, we will start with fitting the simplest possible curve, a straight line, to our data. The equation of a straight line is:

$$T = C_1 t + C_2$$

where C_1 and C_2 are constants.

If we settle for that as the interpolating curve, Matlab can find the "best" (in the least square sense) values for the constants 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 t is 1.)

And having found the constants 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 curve fitting. In particular, the expression for T above is *linear* in the constants C_1 and C_2 to find. That is unlike for the exponential fit, where the constant B was *inside an exponential*, and that was then *multiplied* by A to boot. If the used curve is linear in terms of the unknown constants, like for polynomials, numerical analysts speak of "linear regression". Like "method of least squares", "linear regression" is another term you should try to remember.

Anyway, let's do the line fit:

```
% degree of a line
deg=1;
% find the constants C1 and C2 of the fitted line
CoefLinFit=polyfit(timeMeasuredN,TempMeasuredN,deg)
```

```
CoefLinFit =
   -6.1106    12.0482
```

```
% interpolate again at t = 0.7
timeDesired=0.7
TempLinFit=polyval(CoefLinFit,timeDesired)
TempExact=TempExactFun(timeDesired)
errLinFit=abs(TempLinFit-TempExact)
```

```
timeDesired =  0.70000
TempLinFit =  7.7708
TempExact =  6.7600
errLinFit =  1.0108
```

OOPS! Worse than interpolation!

```
% let's see the linear fit in a plot
TempLinFitPlot=polyval(CoefLinFit,timePlot);
plot(timeMeasuredN,TempMeasuredN,'ok',...
      timePlot,TempExactPlot,'--k',...
      timePlot,TempLinFitPlot,'y')
legend('Measured','Exact','Linear fit')
title('Least-Square Approximation by a Line')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
```

According to the plot, the line does the best it can. Clearly no straight line could approximate the exact curve in this example well.

```
% print the maximum error
errLinFitMax=max(abs(TempLinFitPlot-TempExactPlot))
```

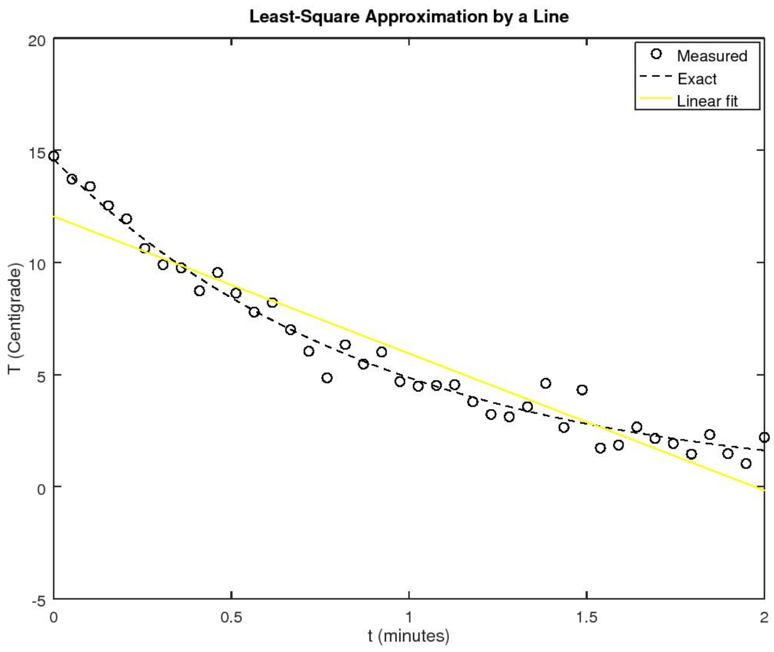
```
errLinFitMax =  2.5518
```

Worse than interpolation, but what do you expect?

```
% try extrapolation again
timeDesired=3
TempLinFit=polyval(CoefLinFit,timeDesired)
TempExact=TempExactFun(timeDesired)
```

```
timeDesired =  3
TempLinFit = -6.2836
TempExact =  0.53849
```

At least that is less crazy than interpolation!



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.

```
% degree of a quadratic (parabola)
deg=2;
% find constants C1, C2, and C3
CoefParFit=polyfit(timeMeasuredN,TempMeasuredN,deg)
```

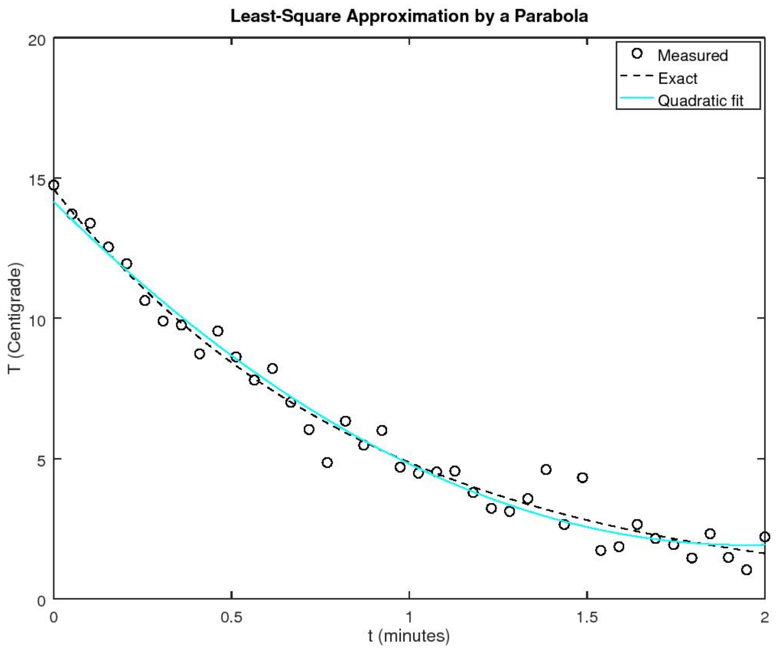
```
CoefParFit =
     3.2292   −12.5690    14.1458
```

```
% interpolate again at t = 0.7
timeDesired=0.7
TempParFit=polyval(CoefParFit,timeDesired)
TempExact=TempExactFun(timeDesired)
errParFit=abs(TempParFit−TempExact)
```

```
timeDesired =  0.70000
TempParFit =  6.9298
TempExact =  6.7600
errParFit =  0.16984
```

That is much better than the interpolations and the linear fit.

```
% let's see the quadratic fit in a plot
TempParFitPlot=polyval(CoefParFit,timePlot);
plot(timeMeasuredN,TempMeasuredN,'ok',...
     timePlot,TempExactPlot,'--k',...
     timePlot,TempParFitPlot,'c')
legend('Measured','Exact','Quadratic fit')
title('Least-Square Approximation by a Parabola')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
```



The plot is a lot better than the previous ones.

```
% print the maximum error
errParFitMax=max(abs(TempParFitPlot-TempExactPlot))
```

```
errParFitMax =   0.45415
```

About 3 times better than interpolation.

```
% try extrapolation again
timeDesired=3
TempParFit=polyval(CoefParFit,timeDesired)
TempExact=TempExactFun(timeDesired)
```

```
timeDesired =   3
TempParFit =   5.5016
TempExact =   0.53849
```

Fitting a quartic

You might think the higher the degree of the polynomial we fit, the better. After all, the higher the degree the more terms in the Taylor series of the exponential can be reproduced. But actually, polynomials of too high order do not work. One big reason is that we are not getting the coefficients of the polynomial from writing down an analytical expression, but from data that have errors in them. If the degree of the polynomial becomes too high, it starts flexing to accommodate the errors.

Think of it. A polynomial of degree 39 has 40 constants. That is enough to make the polynomial go through all 40 inaccurate data points just like the spline and linear interpolation. (And actually, this polynomial would be far, far, far worse than the spline or linear interpolation. Its error would be *many orders of magnitude larger than one.* In fact, that would even be true if you had no errors in the used data at all. High order interpolating polynomials are bad news.)

RULE OF THUMB: *Do not fit using more constants than about the square root of the number of data points. Use even less, if you can get away with it.*

Since we have 40 data points and $\sqrt{40}$ is about 6, we should not fit a polynomial of a degree greater than 5. Actually, since the exact solution is known in this case, you can test what degree produces the smallest error. It turns out that degree 4 and 5 have the same maximum error, but degree 4 has a much better derivative. That illustrates the above point that you want to be as conservative as possible in the number of constants. (On the other hand, the cubic also has about the same error, but a poor derivative.)

So let's fit a polynomial of degree 4, a quartic:

$$T = C_1t^4 + C_2t^3 + C_3t^2 + C_4t + C_5$$

```
% degree of a quartic
deg=4;
```

```
% find the 5 constants
CoefQuartFit=polyfit(timeMeasuredN,TempMeasuredN,deg)
```

```
CoefQuartFit =
    0.68287   -4.11584   10.86547  -17.35302   14.79091
```

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

```
timeDesired =  0.70000
TempQuartFit =  6.7201
TempExact =  6.7600
errQuartFit =  0.039892
```

That is clearly very good.

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

```
% print the maximum error
errQuartFitMax=max(abs(TempQuartFitPlot-TempExactPlot))
```

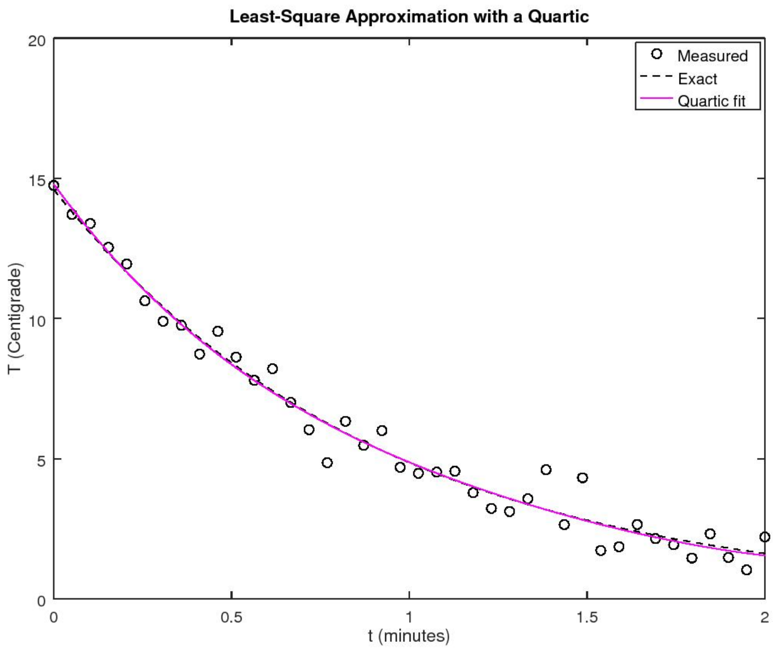
```
errQuartFitMax =  0.19091
```

About 8 times better than interpolation!

```
% try extrapolation again
timeDesired=3
TempQuartFit=polyval(CoefQuartFit,timeDesired)
TempExact=TempExactFun(timeDesired)
```

```
timeDesired =  3
TempQuartFit =  4.7057
TempExact =  0.53849
```

Well, it is a lot less bad than interpolation.



Skip: Fitting an exponential

According to the above, fitting a polynomial of at least quadratic degree worked reasonably 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 not a polynomial 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 constants as

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

this takes the form

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

That is just fitting 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. So we can easily go back and forward between the two.

Below we try this out. Note that Matlab uses `log` for ln (and `log10` for log).

```
% create the measured ln(T) values
lnTempMeasuredN=log(TempMeasuredN);

% a line has degree one
deg=1;
% find C1 and C2
CoefExpFit=polyfit(timeMeasuredN,lnTempMeasuredN,deg)
% the values of A and B that they correspond to
A=exp(CoefExpFit(2))
B=CoefExpFit(1)
```

```
CoefExpFit =
  -1.1490    2.7033
A =  14.929
B = -1.1490
```

Not exactly the same as $A = 14.6$ and $B = -1.1$, but not too bad.

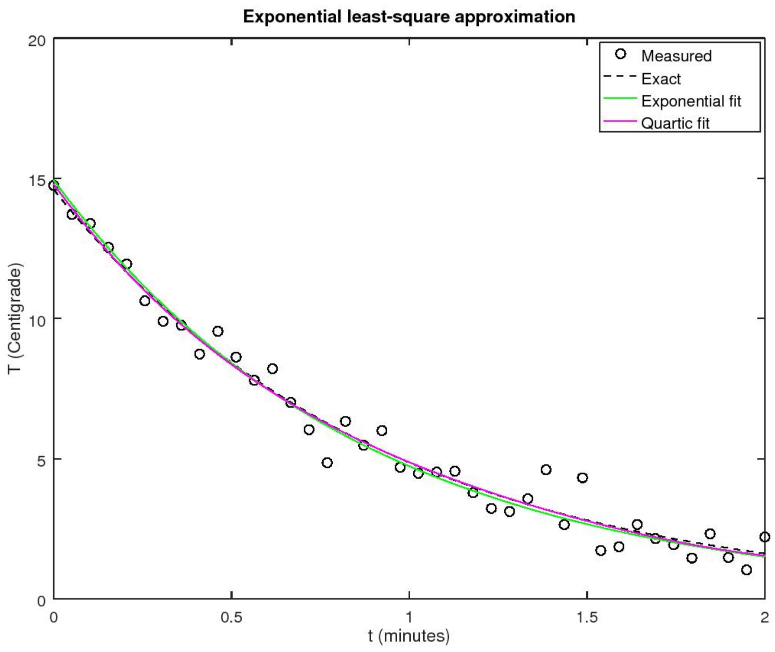
```
% interpolate again at t = 0.7
timeDesired=0.7
% note the exp to convert ln(T) to T
TempExpFit=exp(polyval(CoefExpFit,timeDesired))
TempExact=TempExactFun(timeDesired)
errExpFit=abs(TempExpFit-TempExact)
```

```
timeDesired =  0.70000
TempExpFit =  6.6796
TempExact =  6.7600
errExpFit =  0.080412
```

That is very good.

```
% let's see the exponential fit in a plot
TempExpFitPlot=exp(polyval(CoefExpFit,timePlot));
plot(timeMeasuredN,TempMeasuredN,'ok',...
      timePlot,TempExactPlot,'--k',...
      timePlot,TempExpFitPlot,'g',...
      timePlot,TempQuartFitPlot,'m')
legend(...
     'Measured','Exact','Exponential fit','Quartic fit')
```

```
title('Exponential least-square approximation')
xlabel('t (minutes)')
ylabel('T (Centigrade)')
```



The plot is quite good too. In some ways it is better than the quartic.

```
% print the maximum error
errExpFitMax=max(abs(TempExpFitPlot-TempExactPlot))
% compare with the one of the quartic
errQuartFitMax
```

```
errExpFitMax =   0.32932
errQuartFitMax =   0.19091
```

Good but the quartic had a smaller error.

```
% try extrapolation again
timeDesired=3
TempExpFit=exp(polyval(CoefExpFit,timeDesired))
TempExact=TempExactFun(timeDesired)
```

```
timeDesired =   3
TempExpFit =   0.47542
TempExact =   0.53849
```

This is much better than anything else.

INTEGRALS

It is easy to do determined integrals, with given limits, using Matlab. Just use the `quad` function. (In recent versions of Matlab, from R2012a, you can instead use the `integral` function.)

As an example, we will integrate our temperature between times $t = 0$ and 2, and call the result q:

$$q = \int_{t=0}^{2} T \, \mathrm{d}t$$

If you want some physical meaning for this integral, it is an approximate measure of how much net heat the bar radiates away per unit surface area.

The exact integral using calculus

Finding the exact integral has nothing to do with Matlab. It is pure calculus. But you are supposed to know calculus, so you should be able to do it. The exact temperature was

$$T = 14.6e^{-1.1t}$$

and the antiderivative of that is just the same thing divided by -1.1. And calculus says to subtract the antiderivatives at the limits of integration. That is done below.

```
% the exact integral according to calculus
qExact=TempExactFun(2)/(-1.1)-TempExactFun(0)/(-1.1)
fprintf('Exact integral of the exact temperature: ')
fprintf('%.3f\n',qExact)
```

```
qExact =   11.802
Exact integral of the exact temperature: 11.802
```

Approximate integration of the exact temperature

If we integrate the *exact* temperature *numerically* using `quad` or `integral`, instead of using Calculus, there is going to be some error. Numerical integration is normally not exact. But it is typically very accurate for smooth functions.

Let's try it out for our example. It is easy to use `quad` or `integral`; you just provide it the function to integrate, TempExactFun, and the limits of integration, 0 and 2.

Note: since TempExactFun is not the name of a function, but a handle to an anonymous function, we should not put an @ before it. If you do, Matlab will refuse.

```
% integrate TempExactFun numerically from 0 to 2
qNumExact=quad(TempExactFun,0,2);
% print out the answer and the error in percent neatly
fprintf('Numerical integration: %.3f Error: %.1E%%\n',...
        qNumExact,abs(qNumExact-qExact)/qExact*100)
```

```
Numerical integration: 11.802 Error: 0.0E+00%
```

As this shows, numerical integration of a sufficiently smooth function can be *very* accurate. The error is smaller than the round-off.

Approximate integrals using interpolation or fitting

Remember that the above results cheat. They use the exact temperature that we are not supposed to know. We are only supposed to know the data points and should use those.

Note also that we cannot use the data points directly into `quad` or `integral` because they are just a bunch of numbers, not functions.

However, `interp1`, `spline`, and `polyval` *are* functions, and we can use those.

Unfortunately, `quad` and `integral`, like `fzero` in lesson 2, can only use a function of a *single* input argument. And `interp1`, `spline`, and `polyval` are not just functions of the "desired" time value, but also of other input arguments (the data points or the polynomial constants). So, just like for `fzero` in lesson 2, we will again need to create anonymous functions with single input arguments to give to `quad` or `integral`.

```
% try numerical integration of the linear interpolation
qNumLinear=...
    quad(@(t) interp1(timeMeasured,TempMeasured,t),0,2);
% print out the result and error
fprintf('Linear interpolation:  %.3f Error: %.3f%%\n',...
        qNumLinear,abs(qNumLinear-qExact)/qExact*100)

% try numerical integration of the spline interpolation
qNumSpline=...
   quad(@(t) spline(timeMeasured,TempMeasured,t),0,2);
```

```
% print out the result and error
fprintf('Spline interpolation:  %.3f Error: %.3f%%\n',...
        qNumSpline,abs(qNumSpline-qExact)/qExact*100)

% try numerical integration of the linearN interpolation
qNumLinearN=...
   quad(@(t) interp1(timeMeasuredN,TempMeasuredN,t),0,2);
% print out the result and error
fprintf('LinearN interpolation: %.3f Error: %.3f%%\n',...
        qNumLinearN,abs(qNumLinearN-qExact)/qExact*100)

% try numerical integration of the splineN interpolation
qNumSplineN=...
   quad(@(t) spline(timeMeasuredN,TempMeasuredN,t),0,2);
% print out the result and error
fprintf('SplineN interpolation: %.3f Error: %.3f%%\n',...
        qNumSplineN,abs(qNumSplineN-qExact)/qExact*100)

% try numerical integration of the parabolic fit
qNumParFit=quad(@(t) polyval(CoefParFit,t),0,2);
% print out the result and error
fprintf('Parabolic Fit:         %.3f Error: %.3f%%\n',...
        qNumParFit,abs(qNumParFit-qExact)/qExact*100)

% try numerical integration of the quartic fit
qNumQuartFit=quad(@(t) polyval(CoefQuartFit,t),0,2);
% print out the result and error
fprintf('Quartic Fit:           %.3f Error: %.3f%%\n',...
        qNumQuartFit,abs(qNumQuartFit-qExact)/qExact*100)
```

```
Linear interpolation:  12.095 Error: 2.482%
Spline interpolation:  11.803 Error: 0.011%
LinearN interpolation: 11.745 Error: 0.482%
SplineN interpolation: 11.727 Error: 0.633%
Parabolic Fit:         11.765 Error: 0.315%
Quartic Fit:           11.757 Error: 0.379%
```

As all these examples demonstrate, numerical integration is usually not much of a problem.

DERIVATIVES

Sometimes we are interested in the derivative of the quantity in question.

For example, in the present case the derivative is a measure of how much heat leaks out of the bar per unit volume and time.

Unlike integration, finding derivatives numerically is a tricky business.

The exact derivative using calculus

Finding the exact derivative has nothing to do with Matlab. It is pure calculus. But you are supposed to know calculus, so you should be able to do it. The exact temperature was

$$T = 14.6e^{-1.1t}$$

and the derivative of that is just the same thing multiplied by -1.1.

Since we will want to plot the found derivative against time, we need to evaluate this exact result at the plot points we use.

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

Differentiation of the fitted polynomials

For the linear, quadratic, and quartic fits, we can use the fact that Matlab function `polyder` will find the constants 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 that method to find the derivative of the temperature at our plot points, and then plot the results. We will compare the quadratic and the quartic fits.

Note that the results will be pretty bad:

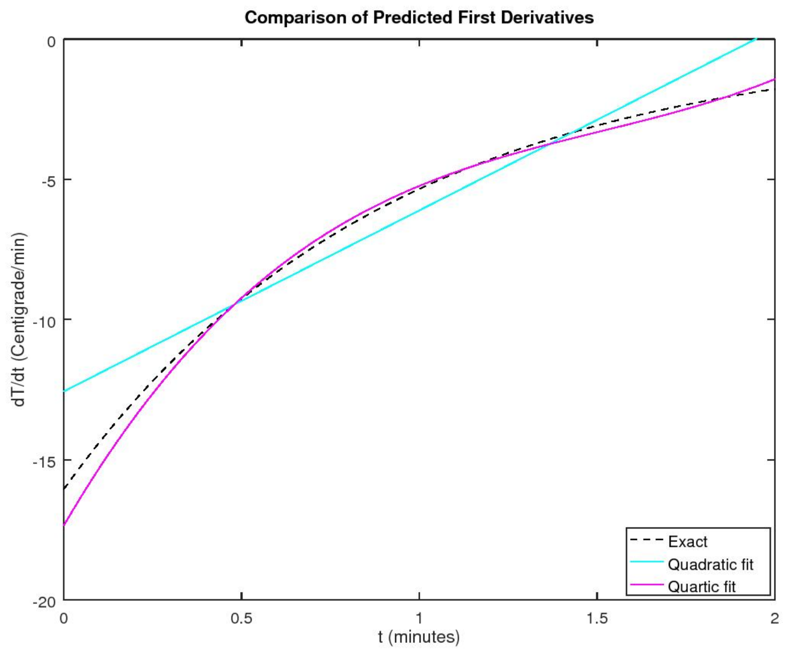
WARNING: *Errors tend to become much worse in derivatives.*

(Additional note: there is also a Matlab function `polyint` that can be used to *integrate* the polynomial fits exactly. But that is much more restricted than `quad` or `integral`; you cannot, say, integrate arbitrary functions of the temperature with `polyint`.)

```
% 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
derCoefQuartFit=polyder(CoefQuartFit);
% use it to evaluate the derivative at the plot points
derTempQuartFitPlot=...
    polyval(derCoefQuartFit,timePlot);
```

```
% plot it
plot(timePlot,derTempExactPlot,'--k',...
      timePlot,derTempParFitPlot,'c',...
      timePlot,derTempQuartFitPlot,'m')
axis([0 2 -20 0])
legend('Exact',...
        'Quadratic fit',...
        'Quartic fit')
legend('location','southeast')
title('Comparison of Predicted First Derivatives')
xlabel('t (minutes)')
ylabel('dT/dt (Centigrade/min)')
```



The polynomial-fit derivatives are pretty bad. The quartic fit is better than the quadratic one, but note the wrong curvature near the end.

Skip: Other ways to do differentiation

One other way to find the derivative of the temperature is to differentiate the fitted exponential. This differentation is done the same way as for the exact temperature above. As might be expected, it gives a better result than the fitted polynomials.

Still another way is to differentiate the interpolated spline. The way to find the derivative of the spline has some rough resemblance to the way we found the derivatives of the fitted polynomials above. In fact, spline interpolation is done by "piecewise polynomials" (pp): there is a different polynomial (a cubic) in each piece between measured points. We can get function `spline` to give us the constants of these polynomials by *not* specifying desired locations to evaluate the spline. Then, in Octave, we can find the derivative polynomials by using function `ppder` (much like `polyder` for single polynomials above). And then we can evaluate that using `ppval` (much like `polyval`).

This is demonstrated below. The derivative of the spline with 5 *exact* measurements is really pretty good. However, the derivative of the spline with 40 *noisy* measurements is very, very bad indeed.

REMEMBER: *Spline differentiation might be good, but noisy data are a big problem.*

If you want to do the below using Matlab instead of Octave, the bad news is that the idiots at MathWorks never defined a function `ppder` to find the derivatives of piecewise polynomials. There are however third-party functions that can do it for you. One example is `ppdiff` in Jonas Lundgren's "SplineFit" package:
Link to Splinefit for ppdiff
A simpler example is Matthew Kelly's `ppDer` (note the capital D):
Link to ppDer
I have not tested them. Let me know if you did.

(Note: Octave also has function `ppint` for exact *integration* of the spline, and so has Splinefit above.)

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

```
% derivative of the exponential fitted to 40 noisy data
derTempExpFitPlot=CoefExpFit(1)*TempExpFitPlot;

% for the spline through the 5 exact measurements:

% piecewise polynomial constants of the spline
ppSpline=spline(timeMeasured,TempMeasured);
% piecewise polynomial constants of the derivative
derppSpline=ppder(ppSpline);
% evaluate at the plot points
derTempSplinePlot=ppval(derppSpline,timePlot);
```

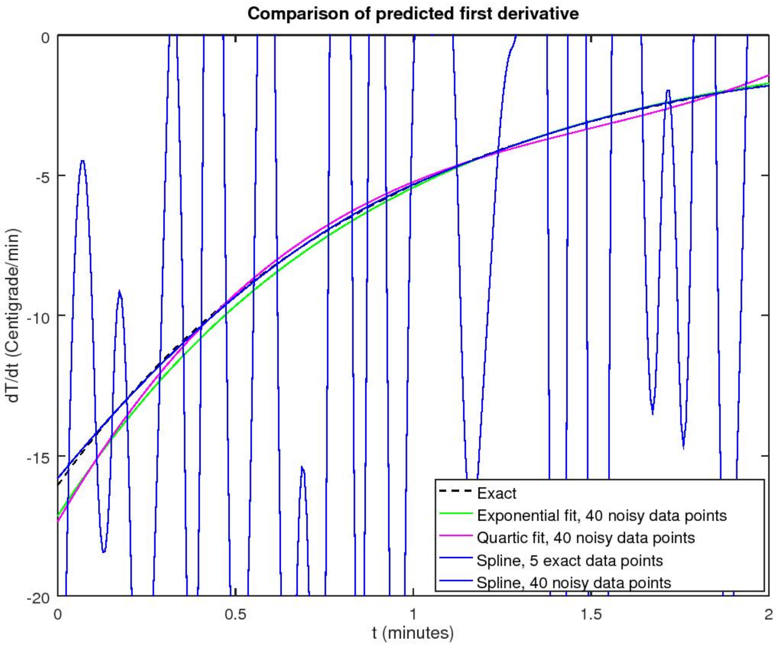
```
% for the spline through the 40 noisy data points:

% piecewise polynomial constants of the spline
ppSplineN=spline(timeMeasuredN,TempMeasuredN);
% find the constants of the derivatives
derppSplineN=ppder(ppSplineN);
% evaluate at the plot points
derTempSplineNPlot=ppval(derppSplineN,timePlot);

% plot it
plot(timePlot,derTempExactPlot,'--k',...
     timePlot,derTempExpFitPlot,'g',...
     timePlot,derTempQuartFitPlot,'m',...
     timePlot,derTempSplinePlot,'b',...
     timePlot,derTempSplineNPlot,'b')
axis([0 2 -20 0])
legend('Exact',...
       'Exponential fit, 40 noisy data points',...
       'Quartic fit, 40 noisy data points',...
       'Spline, 5 exact data points',...
       'Spline, 40 noisy data points')
legend('location','southeast')
title('Comparison of predicted first derivative')
xlabel('t (minutes)')
ylabel('dT/dt (Centigrade/min)')
```

The exponential fit derivative is quite good, better than the quartic fit. And the cubic and quintic fits would be a lot worse than the quartic.

The spline is very good for exact data, but noisy data can be a big problem.


Comparison of predicted first derivative
dT/dt (Centigrade/min)
t (minutes)
0
-5
-10
-15
-20
0
0.5
1
1.5
2
Exact
Exponential fit, 40 noisy data points
Quartic fit, 40 noisy data points
Spline, 5 exact data points
Spline, 40 noisy data points