A Differential Equation for the Juggling Data

Annotated Analyses in Matlab

Before you begin, don't forget to add the path to the functional data analysis functions. On my system, this is achieved by the command

```
addpath('../fdaM')
```

The Data

The data to be analyzed in these notes are the registered X-, Y-, and Z-coordinates for the juggling data described in Chapter 12. It is assumed that you have already registered these data as described in the document "Registering the Juggling Data". If you have done that, and have saved the results as a .mat file juggle, then you just have to enter this command to begin:

```
load juggle
```

The Principal Differential Analysis

We will continue the strategy that we adopted in the notes on registration of keeping the sequences intact. That is, we will develop a differential equation separately for each sequence, and then only combine information across sequences when these equations have been estimated.

The main difference between these analyses and those carried out on the printing data is that here we treat all three coordinates as a unitary system, and develop a single coupled linear differential equation that describes the simultaneous behavior of the three coordinates. By contrast, for the printing data we developed separate equations for each coordinate, and each coordinate equation made no use of any information in the data on the other coordinates.

The function that we use here for the principal differential analysis is `pdasystem`. We will develop a second order linear differential equation in velocity separately for each sequence. We will tell the function that we want a third order equation, but in fact we will weight the position of the coordinates by zero, and only make use of the behavior of velocity, acceleration, and the third derivative, jerk.

Our first task is to set up the arguments for function `pdasystem`. Here we set the order of the equation.

```
pdaorder = 3;
```

The vector `festimate` has three logical values indicating whether we want to estimate the forcing function $_j(t)$ for coordinate *j, j=1, 2, 3*. In this case we will not estimate the forcing functions. This means that the differential equation is *homogeneous*. In this case, we can refer to the residual functions as the forcing functions.

The matrix `westimate` contains logical values indicating which derivative weight functions $_j(t)$ will be estimated and which will be fixed. Only the order zero weight function is fixed. Note that because the sysem is coupled, `westimate` is in effect three 3 by 3 matrices stacked on top of one another, each matrix representing the contributions of a specific derivative to all three coordinates. Moreover, the first of these 3 by 3 matrix is the contribution of the position, or zeroth derivative, and is therefore set to 0 since we will fix these weight functions to zero.

```
festimate = zeros(3,1);
westimate = ones(9,3);
westimate(1:3,:) = 0;
```

We also need to specify the amount of smoothing, if any, that will be applied to the estimates of these functions. In this case we won't be doing any smoothing.

```
flambda = zeros(3,1);
wlambda = zeros(9,3);
```

The intercept and weight functions require a basis specification since the function will return functional data objects for them. Here we use the Fourier basis of with seven basis functions, defined by a constant plus three sine/cosine pairs. We can set the number of basis functions at this point, but the actual basis will have to defined separately for each sequence.

```
nbasispda = 7;
```

Here we set up some arrays to store results.

```
D3mean    = zeros(201,3);
resid     = zeros(201, 3, 123);
residmean = zeros(201,3);
wcoef     = zeros(nbasispda, 9, 3, 10);
```

Now we have to loop through the sequences, with a sizable amount of computation within the loop. Consequently, as we did with the registration process, we will assume that you have defined the loop with these statements

```
m = 0;
for i=1:10
…
end
```

Note that we will need the index variable `m`, and that it is initialized at 0 prior to entering the loop.

Now we look inside the loop. So consider that all the code that we discuss from now on is within the above loop.

First we specify the number of basis functions needed for defining the B-spline expansion of the registered data for this sequence..

```
nbasis  = 1 + 11*seqn(i);
if nbasis  == round(nbasis /2)*2, nbasis  = nbasis  + 1;  end
```

The duration of the sequence must be calculated as well as the duration of the target sequence.

```
period0 = timepercycle*seqn(i);
```

Set up a vector for the range of the sequence.

```
rng0 = [0,period0];
```

Now define a Fourier basis for the observed sequence.

```
basisi = create_fourier_basis(rng0, nbasis, period0);
```

Use this basis as well as the coefficients of the expansion computed in the registration phase to set up a functional data object for the sequence.

```
coefi  = reshape(coefreg(1:nbasis,:,i),[nbasis,1,3]);
fdregi = fd(coefi, basisi);
```

Also set up a basis for the forcing functions and the weight functions.

```
wbasis = create_fourier_basis(rng0, nbasispda, timepercycle);
fbasis = wbasis;
```

Finally, in our setting up of the analysis, we need to supply some default values for the forcing and weight functions. These values won't affect the functions being estimated, but these will supply the zero values for $_{0k}(t)$, as well as for the forcing functions, whic we don't choose to estimate.

```
ffd0 = fd(zeros(nbasispda,3),   fbasis);
wfd0 = fd(zeros(nbasispda,9,3), wbasis);
```

Now we're in a position to complete the principal differential analysis of this sequence.

```
[ffdi, wfdi, resfdi] = pdasystem(fdregi, difeorder, ...
                            fbasis, flambda, ffd0, festimate, ...
                            wbasis, wlambda, wfd0, westimate);
```

Now we need to store some results, both for the entire sequence, and also for each cycle within the sequence. First, store the coefficients of the expansion for the weight functions.

```
wcoef(:,:,:,i) = getcoef(wfdi);
```

This loop passes through each of the cycles and saves the values of the third derivative or jerk and also of the residual function, as well as accumulating the mean jerk and mean residual functions.

```
for j=1:seqn(i)
    m = m + 1;
    timecyclej = timecycle+(j-1)*timepercycle;
    resid(:,:,m) = eval_fd(resfdi,timecyclej);
    D3mat = squeeze(eval_fd(fdregi,timecyclej,3));
    D3mean    = D3mean + D3mat./123;
    residmean = residmean + squeeze(resid(:,:,m))./123;
end
```

This concludes the loop through the ten sequences.

Here is the entire loop.

```
m = 0;
for i=1:10
    fprintf(['Sequence ',num2str(i),'\n'])
    nbasis  = 1 + 11*seqn(i);
    if nbasis  == round(nbasis /2)*2, nbasis  = nbasis  + 1;  end
    period0 = timepercycle*seqn(i);
    rng0 = [0,period0];
    basisi = create_fourier_basis(rng0, nbasis, period0);
    coefi  = reshape(coefreg(1:nbasis,:,i),[nbasis,1,3]);
    fdregi = fd(coefi, basisi);
```

```
    wbasis = create_fourier_basis(rng0, nbasispda, timepercycle);
    fbasis = wbasis;
    ffd0 = fd(zeros(nbasispda,3),   fbasis);
    wfd0 = fd(zeros(nbasispda,9,3), wbasis);
    [ffdi, wfdi, resfdi] = pdasystem(fdregi, difeorder, ...
                           fbasis, flambda, ffd0, festimate, ...
                           wbasis, wlambda, wfd0, westimate);
    wcoef(:,:,:,i) = getcoef(wfdi);
    for j=1:seqn(i)
        m = m + 1;
        timecyclej = timecycle+(j-1)*timepercycle;
        resid(:,:,m) = eval_fd(resfdi,timecyclej);
        D3mat = squeeze(eval_fd(fdregi,timecyclej,3));
        D3mean    = D3mean + D3mat./123;
        residmean = residmean + squeeze(resid(:,:,m))./123;
    end
end
```

Plotting the Results

Now we want to plot up our results. First, let's look at the quality of the fit by plotting the mean residual functions, along with standard error lines; and also with the mean jerk function to provide a point of reference for how the mean residuals are. This is Figure 12.5 in the text.

Here we compute the standard deviations of the residuals functions.

```
residstddev = zeros(201,3);
for m=1:123
    residstddev = residstddev + ...
        (squeeze(resid(:,:,m))-residmean).^2./122;
end
residstddev = sqrt(residstddev)./sqrt(123);
```

Set up some labels for the coordinates.

```
coordlabs = ['X','Y','Z'];
```

Now plot the mean residuals for each variable.

```
for j=1:3
   plot(timecycle,residmean(:,j),                    '-',   ...
        timecycle,residmean(:,j)+2.*residstddev(:,j),'r--', ...
        timecycle,residmean(:,j)-2.*residstddev(:,j),'r--', ...
        timecycle,D3mean(:,j),'--', [0,timepercycle],[0,0],':')
   title(['\fontsize{12}', coordlabs(j),' Coordinate'])
   axis([0,timepercycle,-400,500])
   pause
end
```

We would now like to have a look at the weight functions that we have estimated. We first compute the mean weight function coefficients.

```
wcoefmean = zeros([nbasispda,9,3]);
for i=1:10
    wcoefmean = wcoefmean + wcoef(:,:,:,i)./10;
end
```

Now set up a basis for the weight functions that ranges only over a single cycle, and make a functional data object using the mean weight coefficients.

```
pdabasis = create_fourier_basis([0,timepercycle], nbasispda);
wfd = fd(wcoefmean,pdabasis);
```

Now we plot the weight functions. There will be six panels arranged in two rows and three columns. In the top row are the weights placed on the velocities, and in the bottom row the weights on the accelerations. The first column contains the weights for the X-coordinate, the second for the Y-coordinate, and the last for the Z-coordinate. Within each panel, we have the actual weight functions specific to the derivative and coordinate for the corresponding row and column, respectively. There are three weight functions within each panel, the blue being the weight for the X-coordinate, green for Y-coordinate, and red for the Z-coordinate. You will tend to see the weight being the largest for the coordinate defined by the column. The weights for acceleration are much less than those for velocity because acceleration itself is generally much larger than velocity.

```
wfdmat = eval_fd(wfd,timecycle);
for j=1:3
    subplot(2,3,j)
    temp = squeeze(wfdmat(:,4:6,j));
    plot(timecycle, temp(:,1), '-', ...
         timecycle, temp(:,2), '--', ...
         timecycle, temp(:,3), '-.', ...,
         rng, [0,0], 'k:')
    axis([0, .711, -300, 900])
    if j==1, ylabel('\fontsize{16} Velocity'); end
    title(['\fontsize{16} ',coordlabs(j),' Coordinate'])
    subplot(2,3,j+3)
    temp = squeeze(wfdmat(:,7:9,j));
    plot(timecycle, temp(:,1), '-', ...
         timecycle, temp(:,2), '--', ...
         timecycle, temp(:,3), '-.', ...,
         rng, [0,0], 'k:')
    axis([0, .711, -25, 25])
    if j==1, ylabel('\fontsize{16} Acceleration'); end
end
```

Solving the Differential Equation to Recover Fit to Position

The residual or, equivalently, forcing functions plotted above indicate the fit of the equation to the third derivative. But of course we'd also like to examine the fit of the equation to position, velocity, and acceleration at the same time. That is the great attraction of a differential equation; it gives a model for four levels of derivative rather than just only for the zeroth order derivative, or positon.

The following code solves the homogeneous linear differential equation defined by the weight functions. It uses Matlab function `ode45` and also the functional data analysis function `derivsn`.

```
global wfd
ystart = eye(9);
yarray = zeros(201,9,9);
for i=1:9
  [tp,yarray(:,:,i)] = ode45('derivsn', timecycle, ystart(:,i));
end
```

The array `yarray` contains position values for three linearly independent basis functions spaning the nullspace of the linear differential operator in the first three levels of the second dimension, velocity values in the second three levels, and acceleration values in the last three levels. The dimension of the null space is actually 9, but because we didn't estimate the position weights, we only need the last six basis functions. Here we extract what we need.

```
vel = zeros(201,6,3);
vel(:,:,1) = squeeze(yarray(:,4,4:9));
vel(:,:,2) = squeeze(yarray(:,5,4:9));
vel(:,:,3) = squeeze(yarray(:,6,4:9));
```

Now we can examine each cycle in turn, plotting the observed velocity as a set of points, and the fitted velocity as a solid line.

```
for i=1:10
    nbasis   = 1+11*seqn(i);
    if nbasis == round(nbasis/2)*2, nbasis = nbasis + 1;  end
    period0  = timepercycle*seqn(i);
    rng0     = [0,period0];
    coefi    = reshape(coefreg(1:nbasis,:,i),[nbasis,1,3]);
    basisi   = create_fourier_basis(rng0, nbasis, period0);
    fdregi   = fd(coefi, basisi);
    for j=1:seqn(i)
        timecyclej = timecycle+(j-1)*timepercycle;
        D1y = squeeze(eval_fd(fdregi,timecyclej,1));
        for k=1:3
            zmat = [onen,squeeze(vel(:,:,k))];
            D1yhat(:,k) = zmat * (zmat\D1y(:,k));
        end
```

```
        for k=1:3
            subplot(3,1,k)
            plot(timecycle, D1y(:,k),    '-', ...
                 timecycle, D1yhat(:,k), '--', ...
                 [0,timepercycle], [0,0], ':')
            axis([0,.717,-2.5,2.5])
            ylabel(['\fontsize{16} D',coordlabs(k),' (m/s)'])
            if k==1
                title(['Sequence ',num2str(i), ...
                       ' Cycle ',num2str(j)])
            else
                title('')
            end
        end
        pause
    end
end
```

As you page through these 123 plots, you will perhaps by impressed by how well the solution to the differential equation can track the cycle-to-cycle variation in the velocities. This has been achieved by replacing 369 within-cycle coordinate functions, each requiring about 11 basis functions to represent, by 18 weight functions, each represented by seven basis functions.