MATLAB Functions for Profiled Estimation of Differential Equations

Giles Hooker

May 26, 2006

Contents

1 Introduction to Profiling

The functions described in this software package are designed to provide estimates of a parameter vector θ defining an ordinary differential equation of the form:

$$Dx(t) = f(t, x(t), \theta). \tag{1}$$

Equation (1) typically represents a vector of such equations for a vector-valued x and f. Let $x_i,\ i = 1, \dots, d$ be the components of x. We assume that we have noisy data, $y_i(t_j) = x_i(t_j) + \epsilon_{ij}$, and wish to use this to recover θ. The measurement times t_j will not be constant across components y_i, and the standard deviation of measurement errors ϵ_{ij} may also vary. This software has been produced to take these aspects of the data into account.

More generally, although D is usually regarded as representing a first-order derivative ($Dx(t) = dx(t)/dt$), it can also be regarded as a vector of derivative operators, one for each component of x, and these need not necessarily be first-order. Using zero'th order derivatives for some components corresponds to a Differential-Algebraic Equation. f may also incorporate derivatives of $x(t)$, and delayed evaluation, $x(t - \delta)$. While the initial presentation of the software here will assume an ordinary differential equation, Section 9 will demonstrate the use of generalized functions in this context.

The estimation procedure has two stages of optimization, which we will label the *inner* and *outer* criteria. For the inner criterion, for each candidate value of θ, we estimate a smooth,$\hat{x}_\theta$, of the data that minimizes the penalized sum of squares:

$$G(x, \theta, \lambda) = \sum_{i=1}^{n} \left\{ \|y_i - x_i(t_i)\|^2 + \lambda_i \int (Dx_i(t) - f_i(t, x(t), \theta))^2 \, dt \right\} \tag{2}$$

Where y_i and t_i are intended to indicate the vector of measured values and measuring times for the ith component. $\|\|^2$ is employed to represent a possibly weighted squared error between the data and the corresponding smooth, giving (2) the interpretation of a penalized sum of squares for the discrepancy between the data y and the fit x. λ is some value that trades-off fidelity to the equations and to the data. We can regard x_theta as an estimate of the trajectory, or path, that the system took, and observe that it allows for some divergence from an exact solution to 1 if this is warranted by the data.

The parameters are then estimated by the outer optimization, θ_λ being chosen to minimize:

$$J(\theta, \lambda) = \sum_{i=1}^{n} \|y_i - x_\theta(t_i)\|^2. \tag{3}$$

the squared distance between the data and the smooth.

Numerically, we approximate $x_{\theta,\lambda}$ by a basis function expansion,

$$x_\theta(t) = \sum_{k=1}^{K} \phi_k(t) c_{k,\theta}$$

with basis functions ϕ_i and coefficients c_i. We also approximate the integral in (2) by a quadrature rule that takes the form of weighted a sum of squares. (2) can now be re-expressed as a non-linear least-squares problem and a Gauss-Newton algorithm employed.

Additionally, we can similarly solve (3) via another Gauss-Newton procedure, making use of the derivative

$$\frac{dJ}{d\theta} = \frac{dJ}{dc}\frac{dc}{d\theta}$$

where

$$\frac{dc}{d\theta} = -\left[\frac{d^2G}{dc^2}\right]^{-1} \frac{d^2G}{dc d\theta}$$

by the Implicit Function Theorem.

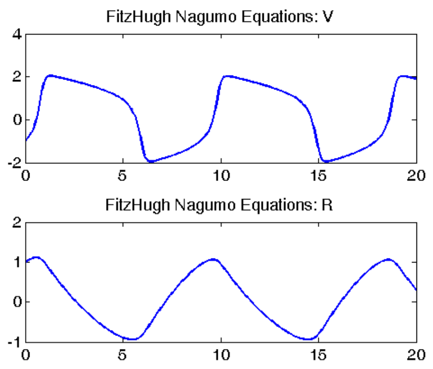
The rest of this document describes how to get this software to do this by making use of the Functional Data Analysis (FDA) software package for MATLAB. A working knowledge of the FDA package is assumed through the rest of the guide.

2 Example: FitzHugh-Nagumo Equations

Throughout this document, we will use the FitzHugh-Nagumo equations as an example. These, relatively simple, equations are given as

$$\dot{x} = c\left(x - \frac{x^3}{3} + y\right)$$
$$\dot{y} = -\frac{1}{c}(x - a + by)$$

with the $\theta = \{a, b, c\}$ being the unknown parameter vector. A plot of solutions to these equations is given below for parameter values $\{0.2, 0.2, 3\}$ and initial conditions $[x_0, y_0] = [-1, 1]$.



3 MATLAB Objects Needed for the Estimation.

3.1 Cell Arrays

The software works with cell arrays throughout with the representation that the contents of one element of an array correspond to one component of the system. These arrays take the form of a *row vector*; one element of the row representing one component of the system. When a system has been run a number of times, replications are represented by the rows of a two-dimensional array. When there are individual numbers which correspond to components of a system – for example, the smoothing parameters, λ – these may be represented by an array rather than a cell-array. The description of the system used here will assume data *without* replications. Section 7 will introduce the modifications necessary to estimate equations with replicated data.

The use of cell arrays is detailed in standard MATLAB manuals, but because of the heavy reliance of this code on them, a quick review of the basics is given below. Cell arrays behave like standard arrays, except that each component may contain an arbitrary MATLAB object. In our case, they will be used to store estimated paths and bases.

Cell arrays are indexed in the same manner as standard arrays. The crucial distinction is that assigning content to arrays makes use of curly braces. Enclosing a vector of objects in curly braces denotes a cell array containing those objects, thus

```
A = cell(1,2);
```

which creates a 1 by 2 array of empty cells, is equivalent to

```
A = {[], []}.
```

Similarly, calling content from a cell array requires curly braces so that

```
A(1,1)
```

returns a 1 by 1 cell array containing an empty object, whereas

```
A{1,1}
```

returns the empty object with `A` defined above. Note that

```
A(1:2)
```

is legitimate notation in MATLAB, but

```
A{1:2}
```

is not. A final shortcut that we make use of is to allow the entries in a cell array to be replicated, so that to insert 0 as the content of both entries of `A` we can either set

```
A(1:2) = {0,0}
```

or

```
A(1:2) = {0}.
```

Note that

```
A{1:2} = 0
```

will produce an error.

3.2 Data Objects

The raw data supplied to the estimation scheme is stored in two cell arrays.

`Tcell` the times at which the component of the system is measured.

`Ycell` the values measured at the times in `Tcell`.

We will use simulated data to demonstrate the system in action. Do do this, we require data in an object `path_cell`, which we set up as follows:

```
Tcell = {0:0.05:20, 0:0.05:20};
path = ode45(@fhnfunode,0:0.05,20,[-1,1],[],[0.2 0.2 3]);
path_cell = {path(:,1), path(:,2)}
```

which produces the paths plotted above. The function `fhnfunode` calculates the right-hand side of the FitzHugh-Nagumo equations for a given vector of inputs and `ode45` is a Runge-Kutta solver for differential equations in MATLAB. From this, we can create data by adding noise to the FitzHugh-Nagumo path:

```
Ycell = path_cell;
for(i = 1:length(path_cell))
  Ycell{i} = path(:,i) + 0.5*randn(size(path,1),1);
end
```

Note that some elements of `Tcell` and `Ycell` may be left as empty cells. These represent unmeasured components of the system. `Tcell` may also be given as a simple vector, in which case all components of the system are assumed to be measured at the same times.

3.3 Basis Objects

We represent the smooth x_j for each component of x by a functional data object stored in a cell array. Each component may be represented by a different basis system. However, it is expected that each basis will cover the same range and will use the same quadrature points. For a cell-array of basis objects, `basis_cell`, the function

```
checkbasis(basis_cell)
```

will verify that all the bases have the same range. This should not be necessary if `basis_cell` is set up as follows.

For ease of use, a function `MakeQuadPoints` is available:

```
quadvals = MakeQuadPoints(knots,nquad)
```

where `knots` is the set of all knots used in B-spline bases across all components of the system and `nquad` is the number of quadrature points to place between knots. This sets up equally spaced quadrature points on these knots and associates Simpson's rule quadrature values with them.

A B-spline basis using these quadrature points can be set up via the function

```
 basis_obj = MakeBasis(range,nbasis,norder,knots,quadvals,nderiv);
```

with the following inputs

range the range of the basis

nbasis the number of basis functions to use

norder the order of the basis functions

knots the knots to use

quadvals as above, quadrature points and values

nderiv the number of derivatives at which a functional data object is expected to be evaluated. This should be the same as the maximum number of derivatives appearing in (1).

We have found that good smooths can require very large numbers of basis functions. However, the order of the B-spline does not seem to affect the quality of the smooth and the minimum value for `nquad`, 5, appears to be sufficient. Usually, only the functional data objects and their first derivatives will need to be evaluated.

The following code sets up basis functions for each of the components in the FitzHugh-Nagumo equation example:

```
knots = 0:0.5:20;
quadvals = MakeQuadPoints(knots,5);

norder = 3;
nbasis = length(knots) + norder - 2;
basis_obj = MakeBasis([0 20],nbasis,norder,knots,quadvals,1);
basis_cell = {basis_obj, basis_obj};
```

3.4 Functional Data Objects

In order to perform the non-linear least squares estimation for the coefficient vectors of these basis functions, initial values need to be provided. They can be set to zero, but it may be advantageous to estimated these by a smooth using a first-derivative penalty. The function `smoothfd_cell` provides a wrapper to `smoothfd` to loop over the values in a cell-array of objects:

```
lambda0 = 0.1;
Lfd_cell = cell(size(basis_cell));

for(i = 1:length(basis_cell))
  fdPar_cell{i} = fdPar(basis_cell{i},1,lambda0);
end

DEfd = smoothfd_cell(Ycell,Tcell,fdPar_cell);
```

When some elements of `Tcell` are empty, the coefficients of that component are estimated as zero. This does not always provide great results and some other initial conditions may be helpful. This might include simply using a nonzero constant. Other possibilities are discussed in Section 5.1.

`DEfd` is now the cell array of functional data objects that we want. For the purposes of smoothing, we need the coefficients of these objects. In this case

```
coefs = getcellcoefs(DEfd);
```

provides these as a single vector concatenated from all the coefficients vectors. This can then be used as an argument to `lsqnonlin` as detailed in Section 5.

3.5 Weights and Smoothing Parameters

Two further objects are needed:

lambda defines the smoothing parameter to use in (2). It may be a vector, defining one parameter for each component of the system. If a singleton, it is assumed to be the same for every component.

wts defines a weight for each observation. It may be empty (each observation gets the same weight), a vector (giving a different weight to each component of the system, but the same weight within a component) or a cell array (defining a different weight for each observation).

In general, `lambda` should be fairly large in order to ensure fidelity to the system, `wts` should be inversely proportional to the size of measurement noise in each component. Alternatively, we might weight by the simple variance in each component:

```
lambda = 1000*ones(size(DEfd));

wts = zeros(size(DEfd));
for(i = 1:length(DEfd))
  wts(i) = 1/sqrt(var(Ycelli));
end
```

4 Defining the Differential Equation

4.1 Derivatives on the Left Hand Side

The left hand side requires a vector `alg` to be specified giving the order of derivative to be used in each component of the system. This is a vector of non-negative integers of the same length as the system, specifying the order of each differential equation.

Usually, as in the case of the FitzHugh-Nagumo equations, this is simply

```
alg = [1 1];
```

but algebraic equations may be specified by setting the corresponding components of `alg` to zero. Higher-order equations may be specified by correspondingly higher entries in `alg`. An example of using algebraic and higher order terms is given in Section 9.

If `alg` is left empty, it is assumed to be a vector of ones.

4.2 Functions for the Right Hand Side

The estimation procedure requires the user to write functions to compute $f(t, x, \theta)$ and several of its derivatives. All these functions take the form

```
fn(t,DEfd,pars)
```

where `t` is a vector of times at which to evaluate the function and `pars` is the vector of parameter estimates. A fourth argument may be specified to contain any extra input into the function. The form of this input is left up to the user, but would typically be a struct object with fields containing additional required quantities.

The output of the function should be a cell-array of values. The cell array will have dimension of #(no. derivatives)+1. These will index the components of F in the first dimension with the derivatives in the following dimensions. Derivatives with respect to components of x will always be taken before derivatives with respect to components of θ. The elements of these cell arrays will be time series corresponding to the evaluation of the relevant component and derivative at the smooth `DEfd` evaluated at times `t`.

To aid in writing these functions, a wrapper function `eval_fdcell` is provided

```
fvals = eval_fdcell(Tcell,DEfd,deriv)
```

where `Tcell` is either a cell-array or a vector (implicitly made into a cell-array all elements containing the vector) of time points at which to evaluate `DEfd` and `deriv` is the order of derivative to take and may be a vector so that

```
eval_fdcell(Tcell,DEfd,0)
```

provides the values of DEfd at the observation times and

```
eval_fdcell(0:20,DEfd,1)
```

provides the first derivatives of DEfd at unit time intervals. The output from these are, of course, cell arrays.

Thus, the function defining the right hand side of the FitzHugh-Nagumo equations will be given by a MATLAB file containing (`p` is substituted for θ throughout the code):

```
function r = fhnfun(t,DEfd,p)

x = eval_fdcell(t,fd_cell,0);
r = x;
r{1} = p(3)*(x{1} - x{1}.^3/3 + x{2});
r{2} = -(x{1} -p(1) + p(2)*x{2})/p(3);

end
```

the derivative of f with respect to the parameters is

```
function r = fhndfdp(t,DEfd,p)

x = eval_fdcell(t,fd_cell,0);
r = cell(2,3);

r(1:2,1:3) = {0};

r{1,3} =  (x{1}-x{1}.^3/3+x{2});
r{2,1} = 1/p(3);
r{2,2} = (-x{2}/p(3));
r{2,3} = ((x{1}-p(1)+p(2)*x{2})/(p(3).^2));

end
```

and the second derivative with respect to x and θ is

```
function r = fhnd3fdxdp(t,DEfd,p)

r = cell(2,2,3);
r(1:2,1:2,1:3) = {0};

r{1,1,3} = 1 - eval_fd(t,fd_cell1).^2;
r{1,2,3} = 1;
r{2,1,3} = 1/p(3)^2;
r{2,2,2} = - 1/p(3);
r{2,2,3} = p(2)/p(3)^2;

end
```

Where a derivative is constant, a simple number can be returned in the corresponding cell and this will save some computation.

In order to perform the profiled estimation scheme, a total of five functions are required:

$$f,\ \frac{df}{dx},\ \frac{df}{d\theta},\ \frac{d^2f}{dx^2},\ \frac{d^2f}{dxd\theta}.$$

If variance estimates are required for the parameters, a further four functions are needed:

$$\frac{d^2f}{d\theta^2},\ \frac{d3f}{dx^3},\ \frac{d3f}{dx^2d\theta},\ \frac{d3f}{dxd\theta^2}.$$

Note that although the examples above are given for an ODE, these functions may also incorporate evaluating derivatives of `DEfd` and evaluating components of `DEfd` at lagged intervals.

The estimation code expects these functions to be given in a struct whose elements are function handles with fields specified in the following manner:

```
fn.fn       = @fhnfun;       % RHS function
fn.dfdx     = @fhndfdx;      % Derivative wrt inputs (Jacobian)
fn.dfdp     = @fhndfdp;      % Dervative wrt parameters
fn.d2fdx2   = @fhnd2fdx2;    % Hessian wrt inputs
fn.d2fdxdp  = @fhnd2fdxdp;   % Hessian wrt inputs and parameters
fn.d2fdp2   = @fhnd2fdp2;    % Hessian wrt parameters.
fn.d3fdx3   = @fhnd3fdx3;    % 3rd derivative wrt inputs.
fn.d3fdx2dp = @fhnd3fdx2dp;  % 3rd derivative wrt intputs and pars.
fn.d3fdxdp2 = @fhnd3fdxdp2;  % 3rd derivative wrt inputs and pars.
                             % dimensions = time, component, input,
                             % parameters
```

and the struct `fn` can now be used as an input into any of the estimating functions.

5 Calling Estimation Functions

The software carries out two tasks. The inner optimization of $G(x, \theta, \lambda)$ defined in ??spline), equivalent to conducting a model-based smooth, and an outer optimization $J(\theta, \lambda)$, or choosing the parameters that optimize the smooth.

5.1 Model-Based Smoothing

The set of coefficients minimizing (2) can be obtained by a call to the MATLAB routine `lsqnonlin` to optimize

```
SplineCoefErr(coefs,basis_cell,Ycell,Tcell,wts,lambda,...
    fn,alg,pars,fn_extras)
```

Here `SplineCoefErr` calculates the value of $G(x, \theta, \lambda)$, along with its derivative with respect to the coefficients defining the smooth x_θ.

`coefs` may be obtained as in §3.3 and `pars` are the parameters θ. `fn_extras` is an optional extra argument to the right hand side its derivatives and does not needed to be included in the function call. All other inputs are as given in the above sections

In the case of the FitzHugh-Nagumo example, we would call `lsqnonlin` as follows:

```
coefs = lsqnonlin(@SplineCoefErr,coefs,[],[],[],basis_cell,...
    Ycell,Tcell,loads,lambda,fn,alg,pars);
```

the cell array of functional data objects can then be recovered by

```
DEfd = Make_fdcell(coefs,basis_cell);
```

As an alternative, the function

```
DEfd = SplineEst(fn,Tcell,Ycell,pars,knots_cell,wts,...
     lambda,lambda0,rough_ord,alg,lsopts,DEfd,fn_extras);
```

provides a wrapper for the call to `lsqnonlin`. It defines a basis using the knots specified in `knots_cell` (again, one set of knots per component of x, but this may be a vector which will then be replicated across all components), and creates an initial smooth defined by `lambda0` and the roughness penalty specified by the `Lfd` object `rough_ord`.

`lsopts` are the optimization options to be passed to `lsqnonlin`. If the functional data object cell array `DEfd` is not empty, this is passed directly to `lsqnonlin` without defining a new cell array of bases.

We could alternatively create `DEfd` this smooth by the call:

```
DEfd = SplineEst(fn,Tcell,Ycell,[0.2 0.2 3],...
     0:0.05:20,wts,1000,0.1,1,[],[],[],[]);
```

This routine may not always give good results when unmeasured components are poorly specified, or when there is relatively little data. Section 8 details a function that will estimate unmeasured components from the others, using the differential equation. Possibly the best solution is to smooth the data with the differential equation using a small value of λ and using this as initial conditions with a larger λ. This scheme may need to be iterated a few times to achieve an appropriate amount of smoothing.

5.2 Profiled Estimation

The profiled estimation routine to estimate θ uses its own Gauss-Newton iteration scheme. This allows DEfd to be updated along with pars, providing some computational speedup. The routine is called by

```
[newpars,DEfd] = Profile_GausNewt(pars,lsopts,DEfd,fn,...
        lambda,Ycell,Tcell,wts,alg,lsopts2,fn_extras,...
        pen,dpen,pen_extras);
```

here lsopts and lsopts2 are optimization options to the outer and inner minimization routines respectively. They follow exactly the optimization toolbox options and may be set with the MATLAB optimset command. As in model based smoothing, fn_extras does not needed to be included in the function call if these two arguments are empty.

Finally, it is possible that we may wish to modify the outer criterion to

$$\tilde{J}(\theta, \lambda) = J(\theta, \lambda) + P(\theta)$$

in which $P(\theta)$ regularizes the estimated values of θ. This will be the case if, for instance, θ is high dimensional. This might occur, for example, if they are taken to be coefficients of a basis expansion for a functional parameter. The entries pen, dpen and pen_extras define such penalties on the parameters. pen and dpen should be functions accepting pars and pen_extras and outputting a vector giving the penalty (to be squared) and it's derivative respectively.

This separate Gauss-Newton optimization routine has been employed so that the object DEfd may be updated as the optimization progresses. Whenever we update θ, the coefficients c will also be updated, and their new values can be anticipated by using $dc/d\theta$.
For the FitzHugh-Nagumo equations, the call becomes

```
lsopts_out = optimset('DerivativeCheck','off','Jacobian','on',...
  'Display','iter','MaxIter',maxit0,'TolFun',1e-8,'TolX',1e-10);

lsopts_in = optimset('DerivativeCheck','off','Jacobian','on',...
    'Display','off','MaxIter',maxit1,'TolFun',1e-14,...
    'TolX',1e-14,'JacobMult',@SparseJMfun);

[newpars,newDEfd] = Profile_GausNewt(pars,lsopts_out,DEfd,fn,...
    lambda,Ycell,Tcell,wts,alg,[],[],[],lsopts_in);
```

Note that an initial guess for `pars` is necessary to start the minimization.

6 Covariance Matrices of Parameter Estimates

A covariance matrix may be calculated for the parameter estimates via a δ-method:

$$var(\theta) \approx \frac{d\theta}{dx}^T var(y) \frac{d\theta}{dx}$$

where

$$\frac{d\theta}{dx} = -\left[\frac{d^2J}{d\theta^2}\right]^{-1} \frac{d^2J}{d\theta dY}$$

These two matrices, along with $var(y)$, must be calculated individually.
$d^2J/d\theta^2$ is calculated using the following function:

```
d2Jdp2 = make_d2jdp2(DEfd,fn,Tcell,lambda,pars,alg,wts,...
    Ycell,fn_extras,d2pen,pen_extras)
```

where `d2pen` is a function providing the second derivative of a penalty with respect to parameters. It takes the same arguments as `pen` and `dpen` in `Profile_GausNewt`.

$d^2J/d\theta dx$ is calculated by the following

```
d2Jdpdy = make_d2jdpdY(DEfd,fn,Tcell,lambda,pars,alg,wts,...
   Ycell,fn_extras)
```

and $var(y)$ is a diagonal matrix calculated in

```
 S = make_sigma(DEfd,Tcell,Ycell,ind)
```

where `ind` indicates the method used to calculate the variance of the observational noise. A value of 0 indicates that all components have an individual variance, 1 indicates pooling across replicates but within components, 2 pooling across components within replicates and 3 pooling across all components. These should be chosen according to the system. It is most likely that 0 or 1 will be appropriate – this will be especially true when different components are measured in different units. However, when different components share the same scales and measurement accuracy, using options 2 or 3 will stabilize the variance estimate.

A covariance matrix for the parameter estimates for the FitzHugh-Nagumo equations can now be calculated by

```
d2Fdp2 = make_d2jdp2(newDEfd_cell,fn,Ycell,Tcell,lambda,...
    newpars,alg,wts)
```

```
d2FdpdY = make_d2jdpdy(DEfd,fn,Ycell,Tcell,lambda,newpars,...
    alg,wts);

dpdY = -d2Fdp2\d2FdpdY;

S = make_sigma(DEfd,Tcell,Ycell,0);

Cov = dpdY * S * dpdY'
```

7 Parameter Estimation with Replication

In some cases, more than one time series corresponding to a system of differential equations may be observed. Moreover, it is possible that only some of the parameters will be common to different replications of the system.

Where replicates of the system are measured, the cell arrays, `Tcell`, `Ycell`, and `DEfd` now become cell matrices with rows representing replications and components given in columns. The bases for different replications do not need to share the same range or quadrature points.

In order to estimate parameters for such systems, new functions need to be used for those in §5 and §6. These take the same arguments as their single-replicate counterparts with the additional input of

parind a matrix whose rows give the indices of the entries of the parameter vector that correspond to parameters for each replicate.

The use of `parind` allows some parameters to be shared and others estimated separately. For instance, if, in the FitzHugh-Nagumo equations, parameters a and b were shared between two replicates, but c was not, we would define the following

```
pars = [a b c1 c2];

parind = [1 2 3;
          1 2 4];
```

if `parind` is left empty, the code uses a default that all parameters are common to all replications.

The input `parind` follows `pars` in each of the functions, `SplineCoefErr_rep`, `Profile_GausNewt_rep`, `make_d2jdp2_rep` and `make_d2jdpdy_rep`. These may all be used with single-replicate systems as well. `make_sigma` already incorporates replications.

8 Estimating Starting Values from a Smooth

This section details two functions that will provide estimates for unmeasured components of a system and initial parameter values respectively. We assume that all components have been estimated by a smooth of the data using, say, a first-derivative penalty.

8.1 Unmeasured Components

Suppose that we have derived `DEfd` from a call to `smoothfd_cell` which has set some unmeasured components to be zero. If we desire a better initial estimate, we could treat the smooths for the measured components of `DEfd` as fixed, and then try to find coefficients for the unmeasured components that best fit the differential equation.

The following function can be optimized with `lsqnonlin`:

```
SplineCoefErr_DEfit(coefs,DEfd,ind,fn,pars,alg,fn_extras)
```

Here `ind` gives the indices of the unmeasured components, `coefs` is a single vector giving initial estimates of the coefficients for the components listed in `ind`. `DEfd` is the fit from the smooth, `pars` are guesses at the parameters and `fn`, `alg` and `fn_extras` are given by the same objects as throughout the rest of the software.

The call to `lsqnonlin` then looks like

```
coefs1 = lsqnonlin(@SplineCoefErr_DEfit,coefs,[],[],[],...
   DEfd,ind,fn,pars,alg,fn_extras);
```

The smooth `DEfd` can then be updated with the call

```
DEfd = update_fdcell(coefs1,ind,DEfd);
```

which replaces the coefficients of the components in `ind` with the estimated `coefs1`. Note here that we assume a knowledge of `pars`, usually as an initial guess. Such an estimate should then be used directly in profile estimation, rather than being re-estimated using the routine below.

8.2 Initial Estimates for θ

An alternative methodology for estimating parameters in differential equations is to first produce a smooth of the data, treat this as fixed, and then choose the parameters that make that smooth look most like a solution. This has the advantage that we are only optimizing over the parameter values, rather than the coefficients, and does not require repeated numerical solutions to the differential equation. Unfortunately, when there is little or noisy data, the smooth produced can be a very poor representation of a differential equation trajectory, especially on the derivative scale. This can lead to highly biassed parameter estimates. Nonetheless, it may be useful to use this technique to obtain initial parameter values from which to start a profiled estimation.

We assume that a smooth `DEfd` to the data has been produced through a call to `smoothfd_cell`. In particular, all components of `DEfd` need to have been measured. The function

```
SplineParsErr(pars,DEfd,fn,fn_extras)
```

can then be used as an argument to `lsqnonlin` to produce parameter estimates. This already requires some initial guess at `pars`, and it may be most useful to simply employ that in profiled estimation.

9 An Example of Generality

So far, the discussion of this software has been given in terms of first-order ordinary differential equations. Here, we give an example of a differential-algebraic equation with delays. Let us take a toy equation as an example:

$$\begin{aligned}\ddot{x}(t) &= ax(t)^2 + by(t)\\ y(t) &= \frac{1}{e^{cy(t)} + x(t)}\end{aligned}$$

This example is intended for expository purposes and is not intended to be realistic.

Here $p = \{a, b, c, d\}$ are considered unknown. This equation can be converted into a single-component system by solving for y at each time t. Doing so is computationally expensive, however, and we can estimate the system directly using the formulation above.

In order to set up the differential equation for the system we first observe that the derivatives on the left hand side correspond to

```
alg = [2 0];
```

We can then define a right hand side function

```
function r = DIFEfun(t,DEfd,p)

x = eval_fdcell(t,DEfd,0);

r = x;

r1 = p(1)*x{1}.^2 + p(2)*x{2};
r2 = 1/(exp(p(3)^x{1})+x{2});

end
```

and derivatives can be taken with respect to this function as normal.

Right hand side functions involving derivative terms of the form

$$\ddot{x}(t) = a\dot{x}(t)^2 + bx(t)$$

currently need to be handled by expanding the system by defining a new variable $y(t) = \dot{x}(t)$ producing

$$\dot{y}(t) = ay(t)^2 + bx(t)$$
$$\dot{x}(t) = y(t).$$

Delay parameters may currently be incorporated only in forcing components. Consider the system:

$$\dot{x}(t) = ax(t) + bf(t-d)$$

the derivative of the right hand side with respect to d is

$$-b\frac{df}{dt}(t-d)$$

so that forcing components must be differentiable. They need to be twice-differentiable in order to accommodate interval estimation.

The software does not currently support delay parameters occurring within components of the system. It also does not support derivatives occurring in the right hand side, except when expanded as suggested above.

10 Forcing Functions and Diagnostics

There are a number of diagnostics that can be used to check the fit of the equations. Among these are the discrepancy between the smooth of the data and an exact solution to the differential equations. This will provide a general indication of regions in which the equations do not hold. A reasonable choice of exact solution would be to begin at the first observation point:

```
smooth = cell2mat(eval_fdcell(0:0.05:20,DEfd));
new_path = ode45(0:0.05:20,odefn,smooth(1,:),[],newpars);

for(i in 1:size(smooth,2))
   subplot(size(smooth,2),1,i)
   plot(0:0.05:20,smooth(:,i),'b')
   plot(0:0.05:20,new_path(:,i),'r')
   plot(Tcell{i},Ycell{i},'g.')
end
```

However, the discrepancy of the result will not generally provide a good indication of how the right hand side may be changed to make the fit better. This is best done by estimating external forcing functions that will make the differential equation fit the data. Adding

$$\dot{x} - f(x,t,\theta)$$

to the right hand side makes the differential equation exact for an estimated x. However, this diagnostic is likely to be biassed since x is already smoothed to be close to an exact solution. This diagnostic also unavailable for systems with unobserved components.

Rather, once an estimate of θ is arrived at, we need to estimate a forcing function that will create a smooth that will make x fit both the (forced) differential equation and the data well. This forcing function can then be plotted against the fitted paths x, derivatives of those paths or external factors. Decomposition techniques such as Independent Components Analysis or functional Principle Components Analysis may also provide useful insights into how the differential equation should be modified.

Such a forcing function may be estimated by expressing it as a basis expansion and treating its coefficients as parameters to be estimated in the profiling scheme already shown. In doing this, the right hand side of the differential equation, including parameters, should remain fixed. This is because any change in the right hand side can be compensated for by changing the forcing functions accordingly.

10.1 Forcing Linear Systems

Where the original differential equations are linear, the profiling procedure can be solved as a linear system. The following function will do this:

```
[smooths,forces] = linforceest((basis_cell,pbasis_cell,A,...
  whichindex,lambda,lambdap,f_lfd,Tcell,Ycell,wts,force,...
  force_extra)
```

The inputs follow the usual conventions, with the following new entries:

basis_cell A cell array of basis objects for representing solutions to the forced differential equation.

pbasis_cell A cell arrray of basis objects for representing forcing functions.

A The matrix in the differential equation

$$Dx = Ax + f$$

whichindex A vector giving the indeces of x which should be assumed to be forced. If this is not specified, it is assumed to be the first index up to the number of entries in `pbasis_cell`

lambdap A penalty parameter for a roughness penalty on the estimated forcing functions. Should normally be zero.

f_lfd The linear differential penalty for penalizing the forcing function.

force Already known forcing components, given as a handle to a function that accepts a vector of times and possibly one further argument and returns the value of the forcing component at the times specified.

force_extra An optional extra component to be input into `force`.

The result of the function call are two cell arrays of functional data objects:

smooths represents the smooths to the data.

forces the estimated forcing components.

10.2 Forcing General Differential Equations

Where the already estimated differential equation is non-linear, however, a Gauss-Newton scheme must be employed as before. For this situation, we regard the coefficients in the basis expansion as parameters and proceed with the usual profiled estimation scheme. In order to facilitate this, the following right-hand side functions have already been written

```
fn.fn       = @forcingfun;       % RHS function
fn.dfdx     = @forcingdfdx;      % Derivative wrt inputs (Jacobian)
fn.dfdp     = @forcingdfdp;      % Dervative wrt parameters
fn.d2fdx2   = @forcingd2fdx2;    % Hessian wrt inputs
fn.d2fdxdp  = @forcingd2fdxdp;   % Hessian wrt inputs and parameters
fn.d2fdp2   = @forcingd2fdp2;    % Hessian wrt parameters.
fn.d3fdx3   = @forcingd3fdx3;    % 3rd derivative wrt inputs.
fn.d3fdx2dp = @forcingd3fdx2dp;  % 3rd derivative wrt intputs and pars.
fn.d3fdxdp2 = @forcingd3fdxdp2;  % 3rd derivative wrt inputs and pars.
                                 % dimensions = time, component, input,
                                 % parameters
```

These require `fn_extras` to be specified. This should be a struct with the following fields:

```
fn_extras.fn      = fn;      % Original right hand side function
fn_extras.dfdx    = dfdx;    % Original RHS derivative wrt inputs
fn_extras.d2fdx2  = d2fdx2;  % Original RHS Hession wrt inputs
fn_extras.d3fdx3  = d3fd2x;  % Original third derivative

fh_extra.pars     = pars;    % Parameters to input to original system
fn_extras.extras  = extras;  % Original fn_extras input into fn

fn_extras.basisp  = basisp;  % Basis representation of forcing functions
fn_extras.which   = which;   % Which components will be forced?
```

Forcing functions can then be estimated by the usual call to `Profile_GausNewt`.

```
[coefs,smooths] = Profile_GausNewt(pars,lsopts_out,DEfd,fn,...
     lambda,Ycell,Tcell,wts,alg,lsopts_in,fn_extras,...
     pen,dpen,pen_extras);
```

The forcing components can then be recovered by

```
forces = Make_fdcell(coefs,basisp);
```

Note that it is here where the entries pen, dpen and pen_extras are often used. d2pen is also needed if you are estimating a Hessian matrix for the parameters. These can be supplied as

```
pen   = @forcingpen;
dpen  = @forcingdpen;
d2pen = @forcingd2pen;
pen_extras.basis = basisp;
pen_extras.deg = 2;
pen_extras.lambda = 0.01;
```

which provides a penalty on the squared integral of the pen_extras.deg derivative of the forcing function, with smoothing parameter pen_extras.lambda.

11 Predefined Right Hand Side Systems

11.1 Forced, Linear Systems

Where a differential equation model is not known, and we have sufficient data, it is possible to build a model to represent the data in much the same way that linear models are developed in ordinary least-squares regression. In the case of differential equations, a linear differential equation takes the place of the linear regression model and estimated forcing functions are used as diagnostics in place of residuals. In addition to the autonomous system, there may be known forcing components and these may also be allowed to enter the model linearly. The linear system is then written as

$$\dot{x} = Ax + Bu$$

where u are known inputs. The entries in the matrices A and B then need to be estimated, although some may be known.

Functions to estimate linear differential equations are provided by

```
fn.fn       = @genlinfun;       % RHS function
fn.dfdx     = @genlindfdx;      % Derivative wrt inputs (Jacobian)
fn.dfdp     = @genlindfdp;      % Dervative wrt parameters
fn.d2fdx2   = @genlind2fdx2;    % Hessian wrt inputs
fn.d2fdxdp  = @genlind2fdxdp;   % Hessian wrt inputs and parameters
fn.d2fdp2   = @genlind2fdp2;    % Hessian wrt parameters.
fn.d3fdx3   = @genlind3fdx3;    % 3rd derivative wrt inputs.
fn.d3fdx2dp = @genlind3fdx2dp;  % 3rd derivative wrt intputs and pars.
fn.d3fdxdp2 = @genlind3fdxdp2;  % 3rd derivative wrt inputs and pars.
                                % dimensions = time, component, input,
                                % parameters
```

These may be used directly. However, the `fn_extras` object may be used to alter the estimation scheme. It should be a matlab struct and may contain some of the following entries

fixed entries The following may be used when some of the entries in the matrix defining the differential equation are known and fixed.

- `mat` A matrix representing a default matrix A. Parameters are estimated with respect to this; assumed zero if not present.
- `sub` a two-dimensional array giving the indices of the entries in `fn_extra.mat` to be estimated. Assumed to be all of them.

forcing functions the following specify forcing functions which enter the differential equation linearly, but which may also depend on the parameters.

- `force` should be a cell vector of functions accepting a vector of times t, parameters p and extra input arguments, each should output a vector giving the value of the forcing function at times t. Alternatively, if any element is a functional data object, it is evaluated at times t.
- `force_mat` a default matrix for B – assumed to be zero if not specified. Parameters are estimated with respect to this matrix.
- `force_sub` a two-dimensional array giving the entries in B which correspond to parameters. This is assumed to give the diagonal of B if not specified.
- `force_input` extra input information to the forcing functions. May be specified in any manner.

11.2 Univariate polynomial functions

A final set of functions are provided that allow polynomial right hand side functions to be estimated for single-component systems with forcing functions that enter linearly. These are given by

```
fn.fn       = @polyfun;       % RHS function
fn.dfdx     = @polydfdx;      % Derivative wrt inputs (Jacobian)
fn.dfdp     = @polydfdp;      % Dervative wrt parameters
fn.d2fdx2   = @polyd2fdx2;    % Hessian wrt inputs
fn.d2fdxdp  = @polyd2fdxdp;   % Hessian wrt inputs and parameters
fn.d2fdp2   = @polyd2fdp2;    % Hessian wrt parameters.
fn.d3fdx3   = @polyd3fdx3;    % 3rd derivative wrt inputs.
fn.d3fdx2dp = @polyd3fdx2dp;  % 3rd derivative wrt intputs and pars.
fn.d3fdxdp2 = @polyd3fdxdp2;  % 3rd derivative wrt inputs and pars.
                              % dimensions = time, component, input,
                              % parameters
```

The order of the polynomial is assumed to be the length of the parameter vector minus one, with the final parameter being a co-efficient of the forcing function. Forcing functions are specified in `fn_extras.forcing`. This should be a function which takes in a vector of times and additional object `fn_extras.fs` and outputs a vector of values. If `fn_extras` is not given, the system is assumed to be forced with a constant 1.