Abstract

This toolbox is a subset of SPOT currently used for determining nonlinear models satisfying variaous stability properties from time domain data. We generally have inputs $u(t) \in \mathbb{R}^{n_u}$ and outputs $y(t) \in \mathbb{R}^{n_y}$ for a set of trials. Most of the tools require approximate "states" of the system, $x(t) \in \mathbb{R}^{n_x}$, and the toolbox provides some simple functions for computing state surrogates.

This manual is in *alpha*. It is incomplete and a work in progress. Please report bugs to Mark M. Tobenkin <mmt at mit dot edu>.

1 Introduction

Not yet supported:

- Continuous time models (dealing with the subtleties of derivative estimation).

1.1 Quick Start

The general work flow consists of the following:

1. Create an `ridata` object from input / output data.
2. Generate candidate states if states are not directly measured, and estimate derivatives for CT models.
3. Construct and `ripmodel` structure, choosing model type and order.
4. Add constraints and objectives to the model, sub-selecting representative data.
5. Run the optimization.
6. Simulate the model and compare to validation data.

The following is an example of a DT fitting procedure contained in `spot/rid/simple_dt_example.m`:

```
load simple_dt_example;
data = ridata({Ytrain,Ytest},{Utrain,Utest},{Ttrain,Ttest});
[data,g] = states(data,'time-delay',2,2);

train = trials(data,1);
test = trials(data,2);

model = ripmodel(data,1,[7 1],g); % -- True system outside
                                  %    model class.

%model = lse_obj(model,data); %-- Equation Error.
```

```
dsel = unf_select(data,200);  %-- Representative subset of samples.

model = rie_obj(model,dsel);  %-- Local Robust Identification Error.

model = model.optimize();
val = model.validate(data);
```

2 `ridata` — Time Domain Data

The `ridata` data structure stores time domain data for identification. The structure stores a set of M *trials*. The following table describes the member variables:

Name	Dim.	Desc.
		Input-Output Data
`nu`	scalar	Input dimension, nonnegative integer.
`ny`	scalar	Output dimension, positive integer.
`N`	`1 × M`	Trial lengths, positive integers.
`D`	scalar	The number of samples (equal to `sum(N)`).
`T`	`1 × D`	Concatenated sample times.
`Y`	`ny × D`	Concatenated output samples.
`U`	`nu × D`	Concatenated input samples.
		State Space Data
`nx`	scalar	State dimension, nonnegative integer.
`X`	`nx × D`	State samples, all trials if `nx` non-zero, empty otherwise.
`V`	`nx × D`	Update samples, all trials if `nx` non-zero, empty otherwise.
`dT`	scalar	Scalar. `dT = 0` indiciates a CT model, `dT < 0` unspecified.

At a minimum, each trial consists of a vector of sample times, `T`, a sequence of outputs, `Y`, and a sequence of inputs `U`. Additionally, a trial may have a sequence of state estimates, `X`, and a sequence of *update information*, `V`. For CT models, the update information approximates the derivative of the states, $\frac{d}{dt}x(t)$, and for DT models the update information is imply the state at the next time sample.

The following methods construct an `ridata`:

```
d = ridata(Y,U,T)
d = ridata(Y,U,T,X,V,dT)
```

The scalar `dT` can either be 0, indicating that `V` is the derivative of `X`, or a positive scalar, indicating a DT data set. All arguments other than `dT` can either be a cell array or matricies. If they are matricies the data is assumed to be from a single trial and then the following table relates their dimensions:

`T`	$1 \times N$
`Y`	$n_y \times N$
`U`	$n_u \times N$
`X,V`	$n_x \times N$

If the arguments are cell arrays they must all have identical dimensions, and each entry is considered a single trial. The corresponding cell entries must be related as above, and the same n_u, n_y, n_x must be used for every entry.

2.1 Selecting and Combining Data

There are several methods for refining and combining `ridata` objects.

2.1.1 `merge`

The `merge` function combines two `ridata` objections by taking the union of their trials, so long as `ny,nu` and `nx` are the same for both. Usage:

```
dmerged = merge(d1,d2);
```

where `d1` and `d2` are `ridata` objects.

2.1.2 `trials`

The `trials` function returns an `ridata` with only the specified trials of the original `ridata`.

```
dsub = trials(dorig,trialnos);
```

The argument `dorig` is an `ridata`. The `trialnos` must be an array of positive integers less than `dorig.M`. The result, `dsub`, is an `ridata` with `length(trialnos)` trials corresponding in the indicies in `trialnos`. They will be listed in the order `trialnos(:)`.

2.1.3 `select`

The `select` function allows the user to specify a subset of points across all trials to form a new `ridata`. Each data-point is placed in a separate trial of length `1`.

```
dsel = select(d,sel);
```

Here `d` is an `ridat` and `sel` is an array of indicies between `1` and `d.D`. The result `dsel` is an `ridata` with `length(sel)` trials each containing a single data-point corresponding to the index in `sel`.

2.1.4 `unf_select`

The `unf_select` function returns a fixed number of samples taken from all trials based on a uniform data-space coverage strategy.

```
dsampled = unf_select(d,K);
dsampled = unf_select(d,K,dist);
```

Here `d` is an `ridata` and `K` is a positive integer. The resulding `ridata`, `dsampled`, will have `K` trials of duration 1.

The argument `dist` is a *distance function*. This should be a function of two arguments, say `x` and `A`. `x` is a column vector of the form `[ y ; u]` if states have not been specified and `[y ; u ; x ; v]` if they have. The matrix `A` has columns which are drawn from the same space as `x`. The function should return a row vector which is the distance from `x` to each column of `A`.

The default `dist` is the squared ℓ_2 norm, normalized by the standard deviation of each coordinate of each element.

In general `unf_select` should be applied *after* states have been determined (see Section 2.2).

2.2 Generating State Estimates

There are several methods in the toolbox for providing estimates of the latent state of system. Most of these are simply linear projections of past input-output history.

For CT models, estimating the derivative of a system presents another challenge. In general, numerical differentiation is too noisy. Several smoothing algorithms are provided for easing this task. To specify a smoothing algorithm, there is an options data structure.

```
options.numerical_diff = '...';
options.numerical_diff_arguments = {};
```

2.2.1 Time Delay Embedding

The simplest state estimate for discrete time systems is the "time-delay" embedding, that is taking the last k samples of the output to be the state.

```
[dnew,g] = states(d,'time-delay',ky);
[dnew,g] = states(d,'time-delay',ky,options);
[dnew,g] = states(d,'time-delay',ky,ku,options);
```

Here `d` is an `ridata` object without states. The state of the system becomes:

$$x(t) = \begin{bmatrix} y(t) \\ y(t-1) \\ \vdots \\ y(t-\texttt{ky}) \end{bmatrix}$$

The default is `ky == 0`. The input becomes:

$$u(t) = \begin{bmatrix} u(t+1) \\ u(t) \\ \vdots \\ u(t-\texttt{ku}) \end{bmatrix}$$

The default `ku == 0`. If the `options.feedthrough == 0` then the $u(t+1)$ term is not included. Necessarily this operation reduces the length of the trials in `ridata`.

The returned value `dnew` includes the states and potentially augmented inputs. The variable `g` is a function such that:

$$\texttt{all}(\texttt{dnew.Y}(:,\texttt{i}) == \texttt{g}(\texttt{dnew.X}(:,\texttt{i}),\texttt{dnew.U}(:,\texttt{i}))),$$

i.e. `g` is the known function mapping state to output.

2.2.2 Linear Filter Bank

For both CT and DT one can use a linear filter bank to approximate states:

```
dnew = states(d,'linear-filter',G,muY);
dnew = states(d,'linear-filter',G,muY,H,muU);
dnew = states(d,'linear-filter',...,options);
```

Here `G` and `H` are transfer function objects. The output of `G` will be the states and the output of `H` will be the new inputs. The filter `G` will be applied to the elements of $y(t) - \texttt{muY}$ trial-by-trial. Similarly, `H` will be applied to $u(t) - \texttt{muU}$.

In general, to calculate the updates each trial will be shortened by one sample.

If `G` (resp. `H`) is single-input it will be applied to each output (resp. input) in turn. Otherwise, `G` must have `d.ny` inputs and `H` must have `d.nu` inputs.

2.2.3 Orthogonal Filter Bank

For both CT and DT one can use a linear filter bank to approximate states:

```
[dnew,g] = states(d,'orth-filter',domain,Gpoles);
[dnew,g] = states(d,'orth-filter',domain,Gpoles,muY);
[dnew,g] = states(d,'orth-filter',domain,Gpoles,muY,Hpoles);
[dnew,g] = states(d,'orth-filter',domain,Gpoles,muY,Hpoles,muU);
[dnew,g] = states(d,'orth-filter',...,options);
```

Here `domain` is one of the strings `'CT'` or `'DT'`. The argument `Gpoles` is a list of poles (be sure they are stable for the appropriate time domain!). These poles are used to generate a filter with appropriately orthogonal impulse responses in the frequency domain. Here `muY` is a mean to be subtracted off of the output data (default `0`). Similarly, `Hpoles` and `muU` are used to construct a filter for augmenting the inputs. If unspecified, the input is not augmented.

The `options` structure has the same interpretation as in Section 2.2.2.

The output `dnew` is the new `ridata` and the variable `g` is a function such that:

$$\texttt{all}(\texttt{dnew.Y}(:,\texttt{i}) == \texttt{g}(\texttt{dnew.X}(:,\texttt{i}),\texttt{dnew.U}(:,\texttt{i}))),$$

i.e. `g` is the known function mapping state to output.

3 rimodel

When first instantiated, an `rimodel` provides facilities for selecting a model structure, solving for an optimized fit and then validating and simulating the models response to inputs.

3.1 Functions for Undetermined `rimodel`

3.1.1 `optimize`

After an appropriate objective has been defined, `optimize` determines the optimal model parameters.

```
mdet = optimize(mundet);
```

Here `mundet` is an undetermined model and `mdet` is the model determined as the optimal solution.

3.2 Functions for Determined `rimodel`

3.2.1 `simulate`

The function `simulate` determines the response to an input:

```
[ts,ys,xs] = simulate(model,u,tspan,x0);
```

The simulation function returns `ts`, a vector of $1 \times N$ ascending time-steps, `ys`, a matrix of `ny` $\times N$ output samples, and `xs`, a matrix of `nx` $\times N$ state samples.

The argument `model` is a determined model. The argument `u` is the input. This is a function from a scalar t, with `tspan`$(1) \leq t \leq$ `tspan`(`end`), to vectors of size $1 \times$ `nu` for CT models and an `nu` $\times N$ array for DT models.

The argument `tspan` is either a 1×2 ascending array of time-steps for CT models, or a $1 \times N$ array. For the former, the values returned will be at the time-steps specified by the variable step integrator. For the latter the values returned will be the response at the specified elements of `tspan`. For DT models, the values returned will always be for incrementing time-steps.

Finally, `x0` is the `nx` $\times 1$ initial condition vector.

3.2.2 `validate`

The function `validate` simulates the model on the input and initial conditions for a set of trials.

```
valdata = validate(model,data);
valdata = validate(model,data,options);
```

Here `data` is an `ridata` of `M` trials. The dimension of the input output and state must match that of the model. For each trial, the response of the model to

the identical input and initial condition is computed. These trials are stored sequentially in `valdata`. A common use case is to separate training and validation data sets using the `trials` function of `ridata`.

The options structure supports plotting options. If `options.plot == 1` then a separate figure for each trial is plotted containg several relevant comparisons of the results.

3.3 `ripmodel` — Implicit Polynomial Models

For continuous time (CT) data the `ripmodel` fits models of the form:

$$\frac{d}{dt}e(x(t)) = f(x(t), u(t)), \qquad y(t) = g(x(t), u(t))$$

and in the discrete time (DT) case:

$$e(x(t+1)) = f(x(t), u(t)), \qquad y(t) = g(x(t), u(t)),$$

where $e : \mathbb{R}^{n_x} \mapsto \mathbb{R}^{n_x}$, $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \mapsto \mathbb{R}^{n_x}$ and $g : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \mapsto \mathbb{R}^{n_y}$ are polynomials or rational functions with fixed denominators. For the models to be well posed, the Jacobian of $e(\cdot)$ must be invertible in the CT case, and the function $e(\cdot)$ must be invertible in the DT case.

The `ripmodel` can be constructed as follows:

```
m = ripmodel(data,edeg,fdeg,gdeg);
m = ripmodel(data,edeg,fdeg,gdeg,options);
m = ripmodel(data,edeg,fdeg,gfun);
m = ripmodel(data,edeg,fdeg,gfun,options);
```

The first argument, `data`, informs the model of the domain (CT vs. DT) and dimensions of the problem. Further, these data-points are used to decide on an appropriate rescaling of the input-output and state domains for numerical accuracy. *This data should cover the regions over which you plan to model the dynamics.*

The argument `edeg` is a positive integer, indicating the maximum total degree of $e(x)$ as a polynomial in $x \in \mathbb{R}^{n_x}$. For $f(x, u)$ and $g(x, u)$ two numbers must be specified as the degrees for x and u. Thus `fdeg` and `gdeg` should each be a two integer array. Optionally, $g(x, u)$ can be fixed by passing a function, `gfun`, of two arguments. Only a limited set of operations are supported inside this function (simple arithmetic etc).

To enable fixed deonominator rational models, set `options.rational = 1`. This option will only work when the degree of $e(x)$ and $f(x, u)$ in x are specified to be odd. This is most useful for restricting continuous time models to disallow finite escape, and to enable global stability requirements.

To require global stability, set `options.globally_stable = 1`.

3.3.1 `rie_bound`

3.3.2 `rie_obj`

The `rie_obj` method adds a local Robust Identification Error objective to the objective function:

```
mnew = rie_obj(m,data);
mnew = rie_obj(m,data,w);
```

Here `m` is an `ripmodel` and `data` is an `ridata` object. The returned model `mnew` will have an objective augmented as described below. The local RIE objective at each data-point is weighted by `w` which is $1 \times$ `data.D`. The default weight is 1.

3.3.3 `lse_obj`

The `lse_obj` method adds a least-squares minimization to the objective function. This really should only be used in conjunction with more sophisticated objectives and constraints offered by the toolbox. This is currently a *very* slow way to solve LS problems (recasting them as a SOCP):

```
mnew = lse_obj(m,data);
mnew = lse_obj(m,data,w);
```

Here `m` is an `ripmodel` and `data` is an `ridata` object. The returned model `mnew` will have an objective augmented as described below. `w` is a $1 \times$ `data.D` array of weights which default to 1.

For DT models, the following term is added to the objective:

$$\sqrt{\sum_{i=1}^{D} \mathtt{w}(i)\|e(v_i) - f(x_i, u_i)\|^2}$$

and for CT models, the following:

$$\sqrt{\sum_{i=1}^{D} \mathtt{w}(i) \left\| \frac{\partial e}{\partial x}(x_i)v_i - f(x_i, u_i) \right\|^2}.$$

where (v_i, x_i, u_i) are the corresponding samples in `ridata`. For both domains, if the output map is not fixed, the following will be added to the objective:

$$\sqrt{\sum_{i=1}^{D} \mathtt{w}(i)\|y_i - g(x_i, u_i)\|^2},$$

where again (y_i, u_i) are from the samples in `ridata`.

As an example:

```
mnew = lse_obj(m,select(data,1000));
```

would select 1000 data points from `data` and

4 Thanks

This work is supported by National Science Foundation Grant No. 0835947 and the Los Alamos ISAMI program.

References

[1] Mark M. Tobenkin, I. R. Manchester, J. Wang, A. Megretski and R. TEdrake. *Convex Optimization Approaches to Nonlinear System Identification.*. arXiv:1009.1670v1.