Probabilistic Inference and Learning to Control: `pilco`

Carl Edward Rasmussen and Rowan McAllister

July 19, 2015

Abstract

The `pilco` toolbox is an Octave and Matlab implementation of time series inference and controller optimisation for nonlinear dynamical systems in continuous-state, discrete-time settings. The toolbox is both flexible and extensible. Flexibility is achieved by allowing user specification of which dynamics models, inference procedures and policy functional forms to use. Users may choose to provide their own data of a dynamical system, or generate synthetic data using any of the accompanying 'scenarios' such as the cart-pole system. Extensibility is a result of the modular code structure, separating class types such as controllers, dynamics models, scenarios, each of which are easily added to.

Contents

1 Overview **2**
- 1.1 Toolbox Components 2
- 1.2 Program Flow 2
- 1.3 States 3
  - 1.3.1 State Representation 3
  - 1.3.2 State Variable Ordering 3
  - 1.3.3 State Variable Classes 4

2 Base functions **4**
- 2.1 Loss 5
- 2.2 Simulate 5
- 2.3 Propagate 8
- 2.4 Rollout 9

3 Controllers **11**
- 3.1 Ctrl 11
- 3.2 CtrlNF 15
- 3.3 CtrlBF 18

1 Overview

There are several supporting structures, classes and functions the user should be aware of. Each is discussed in further details in later sections. Many functions will have derivative counterparts, named similarly except with an additional suffix 'd'.

1.1 Toolbox Components

Scenario: Scenarios are computer simulators of dynamical systems. Each scenario represents a particular dynamical system - such as the cart-pole system - and generates synthetic data using the known laws of motion. To begin a simulation, execute the `doit.m` file within a scenario's directory.

Dynamics Model: A dynamics model object is a probabilistic model which is used to make inferences from observed system evolutions and predictions on future dynamics. Each model is either a Gaussian process (GP), or GP state space model (GP-SSM).

Policy: A policy is a deterministic function, parameterised by θ, that maps state distribution s to action u; $\pi : s \times \theta \rightarrow u$. Various functional forms of the policy are available, including linear, trigonometric, Gaussian radial basis functions (RBFs) and combinations thereof.

Controller: A controller is a wrapper object of the policy function. In addition, a controller computes various covariances, derivatives and can filter the sequence of system-state observations.

Cost: The cost function takes a state distribution s and returns a scalar value on the unit interval; $c : s \rightarrow [0, 1]$ The output represents the instantaneous cost (or measure of *undesirability*) per unit of time of the system being in state s. The cost specifies what it is the user wants to achieve, a 0 cost being the most desirable kind of state, and a 1 being the least desired.

1.2 Program Flow

Select policy function form, parameterised by (initially random) θ:

$$\pi : s \times \theta \rightarrow u. \tag{1}$$

FOR each trial n in $[1, 2, ..., N]$:

- **Infer** dynamics given all observed data so far; $p(s^{t+1}|s^t, u^t, \mathcal{D}^{1:n-1})$.
- **Simulate** system from initial state distribution s_0 until final state distribution s_{horizon}, using the dynamics model to make successive predictions from one time step to the next.
- **Evaluate** controller:

  $$f(s_0; \theta) \quad = \quad \sum_{t=0}^{\text{horizon}} \gamma^t \text{cost}(s_t), \tag{2}$$

  where f is the cumulative discounted cost, θ is the policy parameters. Derivative information $\frac{df}{d\theta}$ is also computed.
- **Optimise** policy (via gradient decent):

  $$\theta^* \quad \leftarrow \quad \arg\min_{\theta} f(s_0; \theta). \tag{3}$$

- **Apply** controller and generate more training data $\mathcal{D}^n$. This may be done using an available scenario, or the user's hardware.

1.3 States

1.3.1 State Representation

The *state* of the control system can be expressed in various ways. The simplest representation is the *physical state*, consisting:

1. present position vector $\mathbf{x}^{\mathrm{t}}$,
2. present velocity vector $\dot{\mathbf{x}}^{\mathrm{t}}$.

For the best results, we incorporate all relevant information such that satisfies the Markov assumption. One way the above Markov assumption is violated is with laggy sensors. Under sensor lag, observations reflect the state of the system in the past, during which time the previous action has still been in effect. Thus previous actions become relevant when inferring the current state. So a better state representation when mitigating the effects of sensor-lag would be:

1. previous action vector $\mathbf{u}^{\mathrm{t}-1}$,
2. present position vector $\mathbf{x}^{\mathrm{t}}$,
3. present velocity vector $\dot{\mathbf{x}}^{\mathrm{t}}$.

The above representation is especially important when sensor lag amounts to a significant proportion of the time discretisation. Now consider the case where, in addition to sensor delay, velocity information $\dot{\mathbf{x}}$ is unavailable. In this case, the 2-Markov state representation is helpful, consisting:

1. previous-previous action vector $\mathbf{u}^{\mathrm{t}-2}$,
2. previous position vector $\mathbf{x}^{\mathrm{t}-1}$,
3. previous action vector $\mathbf{u}^{\mathrm{t}-1}$,
4. present position vector $\mathbf{x}^{\mathrm{t}}$.

Intuitively, the above state representation works well because one could estimate any relevant velocity information using finite differences. All aforementioned state representations are possible in the `pilco` toolbox, including general N-Markov representations. Note the ordering of state variables must be chronological.

The *state distribution* is approximated as a multivariate Gaussian over all state variables. We encode the state distribution with a structure:

```
state   % state structure
state.m % state mean vector
state.s % state covariance matrix
```

1.3.2 State Variable Ordering

Each scenario directory contains its own executable `doit.m` file, used to begin simulations. The doit file specifies the *ordering* of the state variables (amongst other things). For example, consider a hypothetical scenario: a 2-Markov cart-pole with an augmented variable. The state variables would be ordered as follows:

```
1  1  1   oou          even older value of u
2  2  2   ox           old cart position
3  3  3   otheta       old angle of the pendulum
```

```
 4  4  4   ou           old value of u
 5  5  5   x            cart position
 6  6  6   theta        angle of the pendulum
 7         v            cart velocity
 8         dtheta       angular velocity
 9         x - ox       relative change in cart position
10         u            force applied to cart
       7   sin(otheta)  sine of old angle of the pendulum
       8   cos(otheta)  cosine of old angle of the pendulum
       9   sin(theta)   sine of angle of the pendulum
      10   cos(theta)   cosine of angle of the pendulum
```

where the green column of numbers are the indices for each state variable in our state (here our state dimensionality is D = 6). The three types of indices always agree the first D numbers. The red column keeps extending past index D, and includes some extra variables that the simulator will use. The blue column, used by the policy, performs some trigonometric operations on some of the state variables, and thus also has an extends past D.

The red and green indices must refer to variables in a chronological order. The chronological order also takes the understanding that action variables at any time `t` occur *after* state variables at time `t`. E.g. index 10 is the final red index.

Typically a **doit** file will only display the red and green indices. The blue indices are specificed implicitly as a function of the green indices and the set of angular variables discussed below.

1.3.3 State Variable Classes

Let us now group the above variables that we used in the above subsection. There exists three different ways of indexing each state variable in Sec. 1.3.2 (red green and blue) and the sets of indices below have been colour coded w.r.t. which columns of indices they refer to.

```
D    = 6;              % dimensionality of the state
E    = 2;              % number of outputs from the dynamics model
U    = 1;              % dimensionality of the action
angi = [3 6];              % indices for variables treated as angles (sin/cos)
augi = [9];                % indices for variables augmented to ode vars
dyni = [1 2 3 4 5 6];      % indices for input into the dynamics model
dyno = [5 6];              % indices of dynamics model output, and loss input
odei = [5 6 7 8];          % indices for the ode solver
poli = [1 2 4 5 7 8 9 10]; % indices for the inputs to the policy
```

The `dyni` and `dyno` variables are currently redundant, since it is always the case that `dyni = 1:D`, and `dyno = D-E+1:D`.

Note the `[7 8 9 10]` indices of `poli` refer to the trigonometric terms of whatever `angi` specified were angle variables.

2 Base functions

The `base` directory contains common files that are always used to train a controller. Evaluation of a controller's parameterisation is with `loss.m` (Sec. 2.1). The controller's parameterisation *loss* is a function of the cumulative cost distribution, provided by `simulate.m` (Sec. 2.2). An important subroutine of `simulate.m` is the `propagate.m` function (Sec. 2.3). The `propagate.m` function computes predictive state distributions at time step $t + 1$ given the state distribution at time t current time.

2.1 Loss

5a ⟨*loss.m* 5a⟩≡

```
function [f, df] = loss(p, s, dyn, ctrl, cost, H, exp, cc_prev, n)
% If the heuristic function 'explore' is not present (or if it is empty) then
% the expected cumulative cost is returned. If 'explore' is present, then the
% loss is a function of the cumulative cost mean and variance of the current
% parameterisation 'cc', and the parameterisation of the previous rollout
% 'cc_prev', and the number of rollout trials remaining 'n'. Derivatives of
% these quantities are computed when desired.
%
% [f, df] = loss(p, s, dyn, ctrl, cost, H, exp, cc_prev, n)
%
% p              policy parameter structure
% s              initial state structure
% dyn            dynamics model object
% ctrl           controlloer object
% cost           cost object
% H              time-steps horizon
% exp            exploration struct
% cc_prev        cumulative (discounted) cost structure of previous rollout
% n              number of trials remaining (inc. current point in time, so n>0)
% f        1x1   loss
% df       1xP   loss derivative wrt policy parameters
%
% Copyright (C) 2015 by Carl Edward Rasmussen and Rowan McAllister 2015-06-01

ctrl.set_policy_p(p);
exploring = exist('explore','var') && ~isempty(explore);
sargs = {s, dyn, ctrl, cost, H};

if nargout == 1 % no derivatives
  [S, A, ~, cc] = simulate(sargs{:});
  if exploring
    f = explore(exp, cc_prev, n, S, A, cc, [], sargs{:});
  else
    f = cc.m;
  end
else % derivatives
  [S, A, ~, cc, dcc] = simulate(sargs{:});
  if exploring
    [f, df] = explore(exp, cc_prev, n, S, A, cc, dcc, sargs{:});
  else
    f = cc.m;
    df = dcc.m;
  end
end
```

2.2 Simulate

The *cumulative cost* of a distribution over states is defined to be the long term (possibly discounted) total cost of starting in a given state distribution and following a policy π up to a time horizon

H. The simulate function returns states struct arrays, action struct arrays, cost struct arrays, the cumulative costs and possibly cumulative cost derivatives wrt the policy parameters.

5b ⟨*simulate.m* 5b⟩≡

```
function [s, a, c, cc, dcc] = simulate(s, dyn, ctrl, cost, H)

% [s, a, c, cc, dcc] = simulate(s, dyn, ctrl, cost, H)
%
% s          .    state structure
%   m        F*1  mean vector
%   s        F*F  covariance matrix
%   ?             possibly other fields representing additional information
% dyn        .    dynamics model object
% ctrl       .    controller object
%   is       .    struct indexing vectorized state distributions variables
%     m      F*1  indices of mean parameters
%     s      F*F  indices of variance parameters
%   np            number of parameters in the policy
%   ns            number of state distribution parameters (means and variances)
%   policy   .    policy structure
%     p      .    policy parameters structure
% cost       .    cost function object
%   cov      @    function returning cost-covariances between two states
%   fcn      @    function returning expected and variance of a states's cost
%   gamma         discount factor
% H               length of prediction horizon
% s (output) H+1  state struct array containing Gaussian state distributions
%   m        F*1  mean vector
%   s        F*F  covariance matrix
% a          H    action struct array containing Gaussian action distributions
%   m        U*1  mean vector
%   s        U*U  covariance matrix
% c          H+1  cost struct array containing Gaussian cost distributions
%   m             mean scalar
%   s             variance scalar
% cc              cumulative (discounted) cost structure
%   m             mean scalar
%   s             variance scalar
% dcc             derivative structure of cc
%   m        1xP  derivative cc-mean wrt policy parameters, same fields as p
%   s        1xP  derivative cc-variance wrt policy parameters,same fields as p
%
% Copyright (C) 2008-2015 Carl Edward Rasmussen, 2015-05-31

global currT; currT=1;
if ~isfield(s,'s'); s.s = zeros(length(s.m)); end
c = cost.fcn(s);                                                    % init c
cc.m = c.m; cc.s = c.s;                                             % init cc
q = s.s;                                                            % init q
gamma = cost.gamma; D = ctrl.D; F = length(s.m);
if nargout < 5 % no derivatives
  for t = 1:H                                         % iterate up to horizon
    ⟨compute state action cost 7a⟩
    if nargout > 3 % then accumulate discounted cost
      ⟨compute cumulative costs 7b⟩
    end
  end
else % do derivatives
```

```
⟨compute cumulative costs and derivatives 7c⟩
end
```

The states, actions and cost distributions are computed by iterating up to the horizon, propagating forward the state distributions and recording the action distributions and computing the cost.

7a ⟨*compute state action cost* 7a⟩≡ (5b)

```
[s(t+1), C, a(t)] = propagate(s(t), dyn, ctrl);   % propagate state forward
c(t+1) = cost.fcn(s(t+1));                        % calc cost distribution
```

The discounted cumulative cost is computed by iteration up to the horizon, propagating the state distribution forward and accumulating the discounted cost

$$\mathsf{f}^{\mathsf{t}} \;=\; \mathsf{f}^{\mathsf{t}-1} + \gamma^{\mathsf{t}} \mathsf{c}^{\mathsf{t}}, \quad \mathsf{t} = 1, \ldots, \mathsf{H}, \quad \text{and} \quad \mathsf{f}^{\mathsf{t}=0} = 0, \tag{4}$$

where c^{t} is the instantaneous cost at time t and γ is the *discount* factor.

7b ⟨*compute cumulative costs* 7b⟩≡ (5b)

```
cc.m = cc.m + gamma^t*c(t+1).m;
cc.s = cc.s + gamma^(2*t)*c(t+1).s;
% cross-covariance terms:
q = C'*[q, s(t+1).s];
for j = 1:t
  J = (j-1)*F+(1:D); % j'th block of q
  cc.s = cc.s + 2*gamma^(j+t-1)*cost.cov(s(j), s(t+1), q(1:D,J)');
end
```

To accumulative costs and derivatives we again iterate up to the horizon, but now we need to keep track of the derivatives using the chain rule, as we move forward in time. When calling `propagated.m` (the derivative counterpart function of ⟨*propagate.m* 8⟩) it computes the state distribution at time t from the state distribution at time $\mathsf{t}-1$ and derivatives wrt both the state and the policy parameters:

$$\mathsf{s}^{\mathsf{t}} \;=\; \mathsf{g}(\mathsf{s}^{\mathsf{t}-1}, \mathsf{p}), \quad \frac{\partial \mathsf{s}^{\mathsf{t}}}{\partial \mathsf{s}^{\mathsf{t}-1}} \quad \text{and} \quad \frac{\partial \mathsf{s}^{\mathsf{t}}}{\partial \mathsf{p}}, \tag{5}$$

where g is the *transition* function, which takes a distribution over states and a policy and returns the distribution over the next state. The derivatives are propagated forward using the chain rule

$$\frac{d\mathsf{s}^{\mathsf{t}}}{d\mathsf{p}} \;=\; \frac{\partial \mathsf{s}^{\mathsf{t}}}{\partial \mathsf{s}^{\mathsf{t}-1}} \frac{d\mathsf{s}^{\mathsf{t}-1}}{d\mathsf{p}} + \frac{\partial \mathsf{s}^{\mathsf{t}}}{\partial \mathsf{p}}, \quad \text{where} \quad \frac{d\mathsf{s}^{\mathsf{t}=0}}{d\mathsf{p}} = \mathbf{0}, \tag{6}$$

which is iterated forward in time in line 11 below. Finally the derivative of the cumulative discounted cost is the discounted cumulative derivatives in line 12 below

$$\frac{d\mathsf{f}^{\mathsf{t}}}{d\mathsf{p}} \;=\; \frac{d\mathsf{f}^{\mathsf{t}-1}}{d\mathsf{p}} + \gamma^{\mathsf{t}} \frac{d\mathsf{c}_{\mathsf{t}}}{d\mathsf{s}^{\mathsf{t}}} \frac{d\mathsf{s}^{\mathsf{t}}}{d\mathsf{p}} \quad \text{where} \quad \frac{d\mathsf{f}^{\mathsf{t}=0}}{d\mathsf{p}} = \mathbf{0} \tag{7}$$

where f^{t} is the cumulative discounted cost up to time t, c^{t} is the instantaneous cost at time t and γ is the discount factor.

7c ⟨*compute cumulative costs and derivatives* 7c⟩≡ (5b)

```
1 is = ctrl.is;
2 ic = unwrap([is.m(1:D),is.s(1:D,1:D)]);   % cost indices, depend on real vars
3 sdp = cell(H+1); sdp{1} = zeros(ctrl.ns,ctrl.np);          % init derivatives
4 qdp = sdp{1}(is.s,:);
5 dp = zeros(2,ctrl.np);               % first row = mean, second row = variance
6 for t = 1:H                                            % iterate up to horizon
7   [s(t+1), C, a(t), dsds, dsdp, dCds, dCdp] = propagated(s(t), dyn, ctrl);
8   [c(t+1), dcds] = cost.fcn(s(t+1));           % cost and derivative wrt state
9   cc.m = cc.m + gamma^t*c(t+1).m;
```

```
10   cc.s = cc.s + gamma^(2*t)*c(t+1).s;
11   sdp{t+1} = dsds*sdp{t} + dsdp;                                    % chain rule
12   dp = dp + cost.gamma^t*(dcds*sdp{t+1}(ic,:));     % TODO verify gamma power
13   % cross-covariance terms:
14   dCdp = transposed(dCdp,C) + transposed(dCds,C)*sdp{t};
15   qdp = prodd([],dCdp,[q,s(t+1).s]) + prodd(C',[qdp;sdp{t+1}(is.s,:)]);
16   q = C'*[q, s(t+1).s];
17   for j = 1:t % cross terms
18     J = (j-1)*F+(1:D); % j'th column-block of q (real vars only)
19     [cov, dcovdsj, dcovdst, dcovdq] = cost.cov(s(j), s(t+1), q(1:D,J)');
20     cc.s = cc.s + 2*gamma^(j+t-1)*cov;
21     dcov = dcovdsj*sdp{j}(ic,:) + dcovdst*sdp{t+1}(ic,:) + ...
22       dcovdq*transposed(qdp(sub2ind2(F,1:D,J),:),D);
23     dp(2,:) = dp(2,:) + 2*gamma^(j+t-1)*dcov;
24   end
25 end
26 dcc.m = dp(1,:);
27 dcc.s = dp(2,:);
```

Two final subtleties consists in firstly, the derivatives of structures wrt to structures are represented as matrices (ie the structures have been vectorised), such that the products in line 11 and 12 are valid. This requires the structure `dcds` in line 12 to be vectorised (noting that `c` is a struct of two scalars). And secondly, note that in eq. (6) the state may contain fields in addition to `s.m` and `s.s`, but in eq. (7) only the fields `s.m` and `s.s` are considered (as the cost cannot depend directly on additional fields), and this is achieved by addressing directly those relevant indices using the predefined `ctrl.is` index structure in line 2.

2.3 Propagate

Propagate predicts the distribution over states at time `t+1` given the distribution over states at time `t`. Predictions are made using a controller object to compute control actions, and then concatenating state and action information for input into a dynamics model.

8 ⟨*propagate.m* 8⟩≡

```
function [s, C, a] = propagate(s, dyn, ctrl)

% Propagate the state distribution one time step forward.
%
% [s, C, a] = propagate(s, dyn, ctrl)
%
% s         .       state structure
%   m       F x 1   mean vector
%   s       F x F   covariance matrix
%   ?               possibly other fields representing additional information
% dyn       .       dynamics model object
%   D               dimension of the physical state
%   E               dimension of predictions from dyn model
%   pred    @       dynamics model function
%   pn      E x 1   log std dev process noise
% ctrl      .       controller object
%   fcn     @       controller function
%   U               dimension of control actions
% C         F x F   inverse input covariance times input-output covariance
% a         .       action structure
%   m       U x 1   mean vector
%   s       U x U   covariance matrix
%
```

```
24 % Copyright (C) 2008-2015 Carl Edward Rasmussen and Rowan McAllister 2015-06-05
25
26 F = length(s.m); D = ctrl.D; E = dyn.E; U = ctrl.U;          % short hand names
27 Dz = F-D;                                   % length of predicted information states
28 i = 1:D;                                % indices of physical state input variables
29 j = D+1:F;                                          % indices of information state
30 k = F + (1:U);                                          % indices of control actions
31 l = max(k) + (1:Dz);                          % indices of predicted information state
32 m = max([k,l]) + (1:E);                                 % indices of predicted states
33 ij = [i j]; ik = [i k]; kl = [k l]; ijkl = [ij kl];                     % short hand
34 o = [ik(end-D+E+1:end) m l];                              % ind. to select next state
35 M = zeros(max(m),1); M(ij) = s.m; S = zeros(max(m)); S(ij,ij) = s.s;     % init
36
37 [M(kl), S(kl,kl), A, s] = ctrl.fcn(s, dyn);      % control signal and inf state
38 q = S(ij,ij)*A; S(ij,kl) = q; S(kl,ij) = q';               % action covariances
39
40 [M(m), S(m,m), B] = dyn.pred(M(ik), S(ik,ik));    % compute distr of next state
41 S(m,m) = S(m,m) + diag(exp(2*dyn.pn));                       % add process noise
42 q = S(ijkl,ik)*B; S(ijkl,m) = q; S(m,ijkl) = q';       % next state covariances
43
44 C = [eye(F) A [eye(F,D) A(:,1:U)]*B];                 % inv input var times cov
45 % C_exact_ZC = [eye(F,D),A(:,U+1:end)]*C_gph;
46
47 s.m = M(o); s.s = (S(o,o)+S(o,o)')/2; C = C(ij,o);          % select next state
48 a.m = M(k); a.s = (S(k,k)+S(k,k)')/2;                           % control action
```

On line 37 the controller outputs the matrix A. A which is the covariance between the input variable (the state, distributed as a Gaussian of mean $s.m$ and variance $s.s$) and the output variable (a concatenation of the action variable and the state filter prediction variables, distributed as a Gaussian of mean $M(kl)$ and variance $S(kl, kl)$) *pre-multiplied* by the inverse variance of the input variable. I.e.

$$A = \mathbb{V}[s^t]^{-1}\mathbb{C}[s^t, \{u^t, s^{t+1}_{F-D+1:F}\}], \tag{8}$$

where s^t is the state at time t, u^t is the action taken at time t, and $s^{t+1}_{F-D+1:F}$ is the predicted filter state variables (whose indexes are are [F-D+1,F]). On line 40 the dynamics model outputs the matrix B. Similarly, B is an input-output covariance matrix pre-multiplied by the inverse of input variance. We have

$$B = \mathbb{V}[\{s^t, u^t\}]^{-1}\mathbb{C}[\{s^t, u^t\}, s^{t+1}], \tag{9}$$

where s^t is the state at time t and u^t is the action taken at time t resulting in state s^{t+1} at time $t+1$. The use of an pre-multiplied variance inverse of the input variable has the nice property that the covariance between one variable and any ancestor is simply computed with the product of the ancestor's variance and all such C terms along the path in between.

2.4 Rollout

We define a sampled trajectory of a system's possible evolution up to horizon H as a *rollout*. The system transitions from (point mass) state s^t to (point mass) state s^{t+1} as the system observes point observations y^t and applies point control signals u^t at each point in time t. If the state struct `s` does not have a variance field (i.e if `s.s` does not exist), then subroutines will assume they are called by `rollout` opposed to `propagate`.

9 ⟨*rollout.m* 9⟩≡

```
function [data, latent, L] = rollout(start, ctrl, H, plant, cost, verb)
% Compute a state trajectory using an ode solver (and any additional dynamics)
% from a particular starting state with either a particular policy or random
% actions.
```

```
%
% [data, latent, L] = rollout(start, ctrl, H, plant, cost, verb)
%
% start         nX x 1  vector containing start state (without controls)
% ctrl                  controller structure
%   fcn         @       function implementing the controller
%   init        @       function initialising controller's filtered state
%   policy              policy structure
%     fcn       @       policy function
%     p                 parameter structure (if empty use random actions)
%     maxU      nU x 1  vector of control input saturation values
% H                     rollout horizon in steps
% plant                 the dynamical system structure
%   augi                (opt) indices for states passed to augment function
%   augment             (opt) augment state using a known mapping
%   constraint          (opt) stop rollout if violated
%   dyno                indices for states passed to cost
%   noise               observation noise
%   odei                indices for states passed to the ode solver
%   poli                indices for states passed to the policy
% cost                  cost object
% verb                  verbosity level
%
% data                 data struct
%   state     H+1xnX   state matrix
%   action     H x nU  action matrix
% L           loss incurred at each timestep (1 by H)
% latent      matrix of latent states (H+1 by nX)
%
% Copyright (C) 2012-2015 Carl Edward Rasmussen and Rowan McAllister 2015-06-05

clear odestep;                              % clear persistent old action function
if isfield(plant,'augment'), augi = plant.augi;              % sort out indices
else plant.augment = @(x)[]; augi = []; end
calc_loss = nargout > 2;
if nargin < 6; verb = 0; end
D = ctrl.D; E = ctrl.E; F = ctrl.F; U = ctrl.U;
N = length(start); odei = plant.odei;
latent = zeros(H+1, N); y = NaN(H+1, N); L = zeros(1, H+1); u = zeros(H, U);
obs_noise = @()(randn(1,E)*chol(plant.noise));

latent(1,:) = start;                                              % initialise
y(1,1:D-E) = latent(1,1:D-E);
y(1,D-E+1:D) = latent(1,D-E+1:D) + obs_noise();     % add noise to observations
if calc_loss; L(1) = cost.fcn(struct('m',latent(1,1:D)')).m; end

s.m = y(1,1:D)'; s = ctrl.reset_filter(s);             % reset filter if exists

for i = 1:H   % ---------------------------------------------------- run ROLLOUT
  % Test constraints and stop rollout if violated ----------------------------
  if isfield(plant,'constraint') && plant.constraint(latent(i,:))
    H = i-1; if verb; disp('state constraints violated...'); end; break;
  end

  % Apply policy -------------------------------------------------------------
  s.m(1:D) = y(i,1:D)'; % receive an observation
  [uzm,~,~,s] = ctrl.fcn(s);
  u(i,:) = uzm(1:U); % action 'u' component of uzm
```

```
  s.m(D+1:F) = uzm(U+1:end); % predicted filter 'zm' component of uzm

  latent(i+1,1:D-E) = [latent(i,U+E+1:D), u(i,:)];
  latent(i+1,odei) = odestep(latent(i,odei), u(i,:), plant);

  y(i+1,1:D-E) = [y(i,U+E+1:D), u(i,:)];
  y(i+1,D-E+1:D) = latent(i+1,D-E+1:D) +obs_noise(); % TODO: add process noise?

  % Compute Cost ------------------------------------------------------------
  if calc_loss; L(i+1) = cost.fcn(struct('m',latent(i+1,1:D)')).m; end
end
if verb; disp(['Trial lasted ',num2str(floor(H)),' steps']); end

data.state = y(1:H+1,:); data.action = u(1:H,:);
latent = latent(1:H+1,:); L = L(1,1:H+1);            % trim any trailing zeros

function xa = augment(x, plant)
xa(plant.odei) = x;
xa(plant.augi) = plant.augment(xa);
```

3 Controllers

Controller classes act as wrappers to policy functions. The superclass **Ctrl.m** defines the common functions a controller object supports, discussed Sec. 3.1. Then follows two sections on child classes that inherit **Ctrl.m**; the 'No Filter' controller **CtrlNF.m** in Sec. 3.2 and the 'Bayes Filter' controller **CtrlBF.m** Sec. 3.3.

3.1 Ctrl

11 ⟨*Ctrl.m* 11⟩≡

```
classdef Ctrl < handle
  %CTRL, controller superclass, which ctrlBF and ctrlNF inherit.
  %
  % Ctrl Properties:
  %   actuate       -  @      (optional) call this function with the action
  %   angi          -         indicies for vars treated as angles (sin/cos rep)
  %   D             -         number of real-world state variables
  %   dyn           -  .      dynamics model object (only used by CTRLBF class)
  %   E             -         number of physical state variables
  %   F             -         number of state variables (real + info vars)
  %   is            -  .      state index structure
  %   np            -         number of policy parameters
  %   ns            -         state parameters (including filter)
  %   on            -  D x D  non-log observation noise
  %   onp           -  D x D  non-log obs. noise (only last E vars non zeros)
  %   poli          -         indicies for the inputs to the policy
  %   policy        -         policy struct
  %   U             -         number of control outputs
  %
  % Ctrl Methods:
  %   build_state_index  -  (private) builds 'is' field to index state-struct
  %   clear_filter       -  clears all filter variables (if they exist)
  %   Ctrl               -  constructor
  %   fcn                -  main function, computes control signal
  %   random_action      -  outputs a random control, independent of state
  %   reset_filter       -  resets state fields {zs,zc,v} to a broad prior
```

```
%   set_dynmodel       -   sets the dynnamics model object
%   set_on             -   sets the N-Markov observations noise
%   set_policy_opt     -   sets the policy optimisation settings
%   set_policy_p       -   sets the policy parameters
%
% See also CTRLBF.M, CTRLNF.M.
% Copyright (C) 2015 by Carl Edward Rasmussen and Rowan McAllister 2015-06-05

properties (SetAccess = private)
  actuate
  angi
  D
  dyn
  E
  F
  is
  np
  ns
  on
  onp
  poli
  policy
  U
end

methods

  % Constructor
  function self = Ctrl(D, E, policy, angi, poli, actuate)
    % CTRL is the super-class controller constructor
    %
    % ctrl = Ctrl(D, E, policy, angi, poli, actuate)
    %
    % INPUTS:
    %   D                  number of state variables
    %                      (or a ctrl object to be copied)
    %   E                  number of predicted state variables
    %   policy     .       policy struct
    %     fcn      @       policy function
    %     maxU             maximum control output magnitudes
    %     opt      .       optimisation structure
    %       fh             figure handle for minimize() to display to
    %       length         how many optimisations steps
    %       method
    %       verbosity
    %   angi               indicies for variables treated as angles
    %   poli               indicies for the inputs to the policy
    %   actuate    @       function to actuate calculated action

    % Special case input:
    % Another controller object might be the first (and only) input.
    % This allows easy translations from one controller type to the next
    if ~isnumeric(D); assert(nargin < 2); ctrl = D;
      D = ctrl.D;
      E = ctrl.E;
      policy = ctrl.policy;
      angi = ctrl.angi;
      poli = ctrl.poli;
```

```
        actuate = ctrl.actuate;
        self.on = ctrl.on;
        self.onp = ctrl.onp;
        self.dyn = ctrl.dyn;
      end

      self.D = D;
      self.E = E;
      self.policy = policy;
      if ~isfield(policy,'opt'); self.policy.opt = ...
          struct('length',-1000,'method','BFGS','MFEPLS',20,'verbosity',3); end
      if ~isfield(self.policy.opt,'fh'); self.policy.opt.fh = 1; end
      self.U = length(policy.maxU);
      self.build_state_index();
      if ~isfield(policy,'type'); policy.type = ''; end
      if isfield(self.policy,'p');
        self.np = length(unwrap(self.policy.p));
      end

      if exist('angi','var'); self.angi = angi; else self.angi = []; end
      if exist('poli','var'); self.poli = poli; else self.poli = 1:self.D; end
      if exist('actuate','var') && ~isempty(actuate); self.actuate=actuate; end
    end

    function [uM,uS,uC,s] = fcn(self, s)
      % CTRL.FCN is the main function to output control. Sub-classes will
      % override this function.
      %
      % [uM, uS, uC, s, duMds, duSds, duCds, dsds, duMdp, duSdp, duCdp, ...
      %   dsdp] = CTRL.FCN(s,propdyn)
      %
      % self          .          controller structure
      %   actuate     @          function to actuate calculated action
      %   angi                   indices of angular variabels
      %   dyn         @          controller's dynamics model (for CtrlBF only)
      %   E                      number of predictive state variables
      %   on          D x D      observation noise
      %   onp         D x D      observation noise (only last E vars non-zero)
      %   poli                   indices of policy input
      %   policy      .          policy structure
      %     fcn       @          policy function
      %   U                      number of control outputs
      % s             .          state structure
      %   m           F x 1      state mean
      %   s           F x F      state variance
      %   v       (F-D) x (F-D) filter variance
      % propdyn       @          propagates's dynmodel (for CtrlBF)
      % M         (U+D) x 1      control signal mean vector
      % S         (U+D) x (U+D) control signal variance matrix
      % C             F x (U+D) input-output covariance matrix
      % dMds      (U+D) x S      derivatives of outputs wrt input state struct
      % dSds    (U+D)^2 x S
      % dCds    F*(U+D) x S
      % dsds          S x S      ouput state derivative wrt input state
      % dMdp      (U+D) x P      P is the total number of parameters is the policy
      % dSdp    (U+D)^2 x P
      % dCdp    F*(U+D) x P
      % dsdp          S x P      ouput state derivative wrt policy parameters
```

```
      %
      % See also CTRLBF.FCN, CTRLNF.FCN.
      if strcmp(self.policy.type, 'random')
        uM = self.random_action();
        uS = zeros(self.U); uC = zeros(self.D,self.U);
      end
    end

    % Filter Clearer
    function s = clear_filter(self, s)
      % Clears all filter variables (if they exist).
      % s = CTRL.CLEAR_FILTER(s)
      %   s:  state struct
      i = 1:self.D;
      s.m = s.m(i);
      if isfield(s,'s'); s.s = s.s(i,i); end
      if isfield(s,'v'); s = rmfield(s,'v'); end
    end

    function u = random_action(self)
      % u = CTRL.RANDOM_ACTION(), outputs a random action
      %   u:   U x 1 control action
      u = self.policy.maxU.*(2*rand(1,self.U)-1);
    end

    function s = reset_filter(~, s)
      % s = CTRL.RESET_FILTER(s), does nothing unless overwritten
      %   s:  state struct
      % See also CTRLBF.RESET_FILTER
    end

    function set_dynmodel(self, dyn)
      % CTRL.SET_DYNMODEL(dyn), for updating controller's dynmodel
      %   dyn: dynamics model object
      % Note: dyn property only ever used by CTRLBF
      self.dyn = dyn;
    end

    function set_on(self, onE)
      % CTRL.SET_ON(onE), sets observation noise
      %   onE: E x 1, log observation noise (physical state-variables only)
      assert(self.E == length(onE));
      onD = [-inf(self.U,1); onE(:)];
      onD = repmat(onD,ceil(self.D/(self.U+self.E)),1);
      onD = onD(end-self.D+1:end);
      self.on = diag(exp(2*onD));
      p = self.D-self.E+1:self.D;
      self.onp = 0*self.on; self.onp(p,p) = self.on(p,p);
    end

    function set_policy_p(self, p)
      % CTRL.SET_POLICY_P(p), updateds the policy parameters
      %    p:   policy parameter struct
      self.policy.p = p;
      self.np = length(unwrap(p));
    end

    function set_policy_opt(self, opt)
```

```
      % CTRL.SET_POLICY_OPT(p), updateds the policy optimisation settings
      %    opt:   policy optimisation settings
      self.policy.opt = opt;
    end

  end

  methods (Access = private)
    ⟨function-build-state-index 15⟩
  end

end
```

PILCO stores derivatives of any object as matrices, where rows represent vectorised dependent variables, and columns represent vectorised independent variables. To vectorise any object we use the `unwrap.m` function. The state distribution is stored as a structure. Thus when handling state derivatives, we need to keep track of which elements in the native structure correspond to which index in the vectorisation of that structure. We do so by using a *state index* structure, which has the exact same form of the state structure, except each element within the index structure is a unique integer, corresponding to that element's index when in vectorised form. The state index is built using ⟨*function-build-state-index* 15⟩:

15 ⟨*function-build-state-index* 15⟩≡ (11)

```
function build_state_index(self)
  % CTRL.BUILD_STATE_INDEX(), computes internal field 'is' - a state struct
  % whose members are a states members' indexes. Requires subclass to
  % implement function CTRL.RESET_FILTER
  s.m = nan(self.D,1);
  s = self.reset_filter(s); % generates possible filter variables
  s.s = nan(length(s.m));
  if isfield(s,'reset'); s = rmfield(s,'reset'); end
  self.ns = length(unwrap(s));
  self.is = rewrap(s,1:self.ns);
  self.F = length(s.m);
end
```

3.2 CtrlNF

The most straightforward controller class is one which inputs (noisy) signals from sensors directly into the policy function. The 'No Filter' controller class `control/CtrlNF` implements exactly this.

The process of controlling a dynamical system using `CtrlNF.m` is depicted in Fig. 1. It is assumed the latent system state X_t is observed by imperfect sensors as Y_t, where the random variable Y_t is equal to X_t plus some Gaussian noise:

$$Y_t \sim X_t + \epsilon_t, \tag{10}$$
$$\epsilon_t \overset{iid}{\sim} \mathcal{N}(0, \Sigma_\epsilon). \tag{11}$$

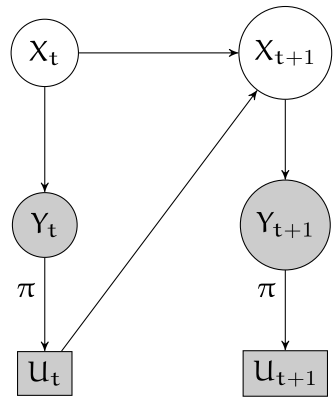

Figure 1: Directed graphical model of a system controller with No Filter from timestep t to timestep $t+1$. A dynamical system begins in state X_t, which sensors observe noisily as Y_t. The controller uses Y_t to decide control signal U_t via policy function π. Finally, the control signal U_t is applied to the system, resulting in new state X_{t+1}.

16a ⟨*CtrlNF.m* 16a⟩≡

```
classdef CtrlNF < Ctrl & handle

  % Controller with No Filter. The state is given either as a point s.m or as a
  % distribution N(s.m,s.s). First augment with trignometric functions if
  % necessary, then call the policy, and finally (optionally) call the actuate
  % function. There is no filter, so no updates to any state structure filter
  % fields are required.
  %
  % See also CTRL.M, CTRLNFT.M.
  % Copyright (C) 2015 by Carl Edward Rasmussen and Rowan McAllister 2015-06-02

  methods

    % Constructor
    function self = CtrlNF(varargin)
      % CTRLNF.CTRLNF is the sub-class constructor of CTRL.CTRL
      % For help, see also Ctrl.Ctrl
      self@Ctrl(varargin{:});                          % Super Ctrl constructor
    end

    % Main function
    function [uM, uS, uC, s, duMds, duSds, duCds, dsds, ...
        duMdp, duSdp, duCdp, dsdp] = fcn(self, s, ~)
      % For help, see also CTRL.FCN

      ⟨initialise 16b⟩
      ⟨augment state with trig variables 17⟩
      ⟨compute control signal 18⟩
    end

  end

end
```

16b ⟨*initialise* 16b⟩≡ (16a)

```
% CtrlNF only operates on a noisy version of the state:
D = self.D; DD = D*D;
```

```
s = self.clear_filter(s);           % clear any filter variables in state
assert(length(s.m) == D);
if isfield(s,'s')
  sy = s.s + self.on;           % propagate mode (distribution of states)
else
  sy = zeros(D);                 % rollout mode (point-mass state sample)
end

angi = self.angi; poli = self.poli; A = length(angi);
derivativesRequested = nargout > 4;
ns = self.ns; is = self.is;
D1 = D + 2*A;
i=1:D;
M = zeros(D1,1); M(i) = s.m; S = zeros(D1); S(i,i) = sy;
if derivativesRequested
  idx = @(i,j,I) bsxfun(@plus, I*(i'-1), j);
  Mds = zeros(D1,ns); Mds(i,is.m) = eye(D);
  Sds = zeros(D1*D1,ns); Sds(:,is.s) = kron(Mds(:,is.m),Mds(:,is.m));
  dsds = zeros(self.ns); dsdp = zeros(ns,self.np);
  dsds(is.m,is.m) = eye(D); dsds(is.s(:),is.s(:)) = eye(D*D);
  dsds(is.s,is.s) = symmetrised(dsds(is.s,is.s),[1,2]);
end
```

The state distribution `s`, is then extended the sine and cosine each angle variable (indexed by `angi`) using `gTrig.m`. The `gTrig.m` function also outputs covariance information C_{gtrig}, which relates to the input-output covariance except with pre-multiplicative input variance:

$$\mathbb{C}[Y_t, Y_t^\circ] = \mathbb{V}[Y_t] C_{gtrig}[Y_t, Y_t^\circ] \tag{12}$$

17 ⟨*augment state with trig variables* 17⟩≡ (16a)

```
% augment state with trig functions
i = 1:D; k = D+1:D1;
if ~derivativesRequested
  [M(k), S(k,k), cg] = gTrig(M(i), S(i,i), angi);
else
  kk = idx(k,k,D1); ik = idx(i,k,D1); ki = idx(k,i,D1);
  [M(k), S(k,k), cg, Mds(k,is.m), Sds(kk,is.m), cgdm, ...
    Mds(k,is.s), Sds(kk,is.s), cgds] = gTrig(M(i), S(i,i), angi);
  qdm = prodd(S(i,i),cgdm);
  Sds(ik,is.m) = qdm; Sds(ki',is.m) = qdm;
  qds = prodd(S(i,i),cgds) + prodd([],'eye',cg);
  Sds(ik,is.s) = qds; Sds(ki',is.s) = qds;
end
q = S(i,i)*cg; S(i,k) = q; S(k,i) = q';
```

The policy, which may be a function of both the regular state variables, and augmented state variables is then computed:

$$\mathsf{U}_t \leftarrow \pi(\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}) \tag{13}$$

The non-trivial part is computing output `uC` which is covariance between state input and control output $\mathbb{C}[\mathsf{X}_t, \mathsf{U}_t]$. We begin noting:

$$\begin{aligned}
\mathbb{C}[\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}, \mathsf{U}_t] &= \mathbb{V}[\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}] C_{\mathsf{poli}}[\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}, \mathsf{U}_t] && (14)\\
\begin{bmatrix} \mathbb{C}[\mathsf{Y}_t, \mathsf{U}_t] \\ \mathbb{C}[\mathsf{Y}_t^\circ, \mathsf{U}_t] \end{bmatrix} &= \begin{bmatrix} \mathbb{V}[\mathsf{Y}_t] & \mathbb{C}[\mathsf{Y}_t, \mathsf{Y}_t^\circ] \\ \mathbb{C}[\mathsf{Y}_t^\circ, \mathsf{Y}_t] & \mathbb{V}[\mathsf{Y}_t^\circ] \end{bmatrix} C_{\mathsf{poli}}[\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}, \mathsf{U}_t] && (15)\\
\therefore \mathbb{C}[\mathsf{Y}_t, \mathsf{U}_t] &= \Big[\mathbb{V}[\mathsf{Y}_t], \mathbb{C}[\mathsf{Y}_t, \mathsf{Y}_t^\circ]\Big] C_{\mathsf{poli}}[\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}, \mathsf{U}_t] && (16)\\
&= \mathbb{V}[\mathsf{Y}_t][I, C_{\mathsf{gtrig}}[\mathsf{Y}_t, \mathsf{Y}_t^\circ]] C_{\mathsf{poli}}[\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}, \mathsf{U}_t] && (17)
\end{aligned}$$

Thus,

$$\begin{aligned}
\mathtt{uC} = \mathbb{C}[\mathsf{X}_t, \mathsf{U}_t] &= \mathbb{C}[\mathsf{X}_t, \mathsf{Y}_t]\mathbb{V}[\mathsf{Y}_t]^{-1}\mathbb{C}[\mathsf{Y}_t, \mathsf{U}_t] && (18)\\
&= \mathbb{V}[\mathsf{X}_t][I, C_{\mathsf{gtrig}}[\mathsf{Y}_t, \mathsf{Y}_t^\circ]] C_{\mathsf{poli}}[\{\mathsf{Y}_t, \mathsf{Y}_t^\circ\}, \mathsf{U}_t] && (19)
\end{aligned}$$

18 ⟨*compute control signal* 18⟩≡ (16a)

```
% compute control signal
if ~derivativesRequested
  [uM, uS, uC] = self.policy.fcn(self.policy, M(poli), S(poli,poli));
else
  [uM, uS, uC, mdm, sdm, cdm, mds, sds, cds, duMdp, duSdp, duCdp] = ...
    self.policy.fcn(self.policy, M(poli), S(poli,poli));
end
if isfield(self, 'actuate'), self.actuate(uM); end   % actuate controller

ec = [eye(D) cg]; ecp = ec(:,poli);
if derivativesRequested
  poli2 = idx(poli,poli,D1); ii = sub2ind2(D1,i,i);
  duMds = mdm*Mds(poli,:) + mds*Sds(poli2,:);
  duSds = sdm*Mds(poli,:) + sds*Sds(poli2,:);

  duC = cdm*Mds(poli,:) + cds*Sds(poli2,:);
  decp = [zeros(DD,ns); cgdm*Mds(i,:) + cgds*Sds(ii,:)];
  decp = decp(sub2ind2(D,1:D,poli),:);
  duCds = prodd(ecp,duC) + prodd([],decp,uC);

  duCdp = prodd(ecp,duCdp);

  duCds(:,is.s) = symmetrised(duCds(:,is.s),2);
  duMds(:,is.s) = symmetrised(duMds(:,is.s),2);
  duSds(:,is.s) = symmetrised(duSds(:,is.s),2);
end
uC = ecp*uC;
```

3.3 CtrlBF

`CtrlBF.m` implements control using Bayesian filtering. The *state* of the control system using filtering contains two parts:

1. the *latent state* x,

2. the *filter state* distribution $b \sim \mathcal{N}(z, V)$.

Thus, the *state distribution* is in principle a distribution over the random variables, x, z and V. However, as an approximation we are going to assume that the distribution on the variance V is just a delta function (ie, that the variance is some fixed value). Assuming further that the state distribution is Gaussian

$$\begin{bmatrix} x \\ z \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m_x \\ m_z \end{bmatrix}, \begin{bmatrix} \Sigma_x & \Sigma_{xz} \\ \Sigma_{zx} & \Sigma_z \end{bmatrix} \right). \tag{20}$$

A Bayes filter (BF) maintains a belief-posterior on x, denoted B_{tt}, conditioned on all information available thus far: the entire history of the system observations $y_{1:t}$ and applied control signals $u_{1:t-1}$. Conditioning on more information than the current observation y_t yields a more informed (and smooth) estimate of x_t. Being a function of all observations, B_{tt} is less susceptible to the noise injected into the most recent observation y_t, and consequently the controller's input is much smoother. To maintain B_{t+1} the BF makes two recursive updates per timestep:

1. Update step: Compute B_{tt} using prior belief $B_t = p(x_t)$ and observation likelihood $\mathcal{L}(x_t|y_t) = p(y_t|x_t)$,

2. Predict step: Compute B_{t+1} by mapping updated belief B_{tt} through transition model $p(x_{t+1}|b_{tt}, u_t)$.

A directed graphical model of a Bayes filter is shown Fig. 3.3: