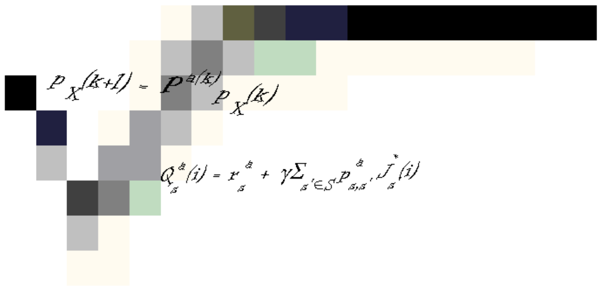
Users' Guide to CFMC-Toolbox

Enso Ikonen

November 30, 2007

Contents



$$p_X^{(k+1)} = P^{a(k)} p_X^{(k)}$$

$$Q_s^a(i) = r_s^a + \gamma \Sigma_{s' \in S} P_{s,s'}^a J_s^*(i)$$

1 CFMC

Let the process under study be described by the following discrete-time dynamic system and measurement equations

$$\mathbf{x}(k) = f(\mathbf{x}(k-1), \mathbf{u}(k-1), \mathbf{w}(k-1)) \quad (1)$$

$$\mathbf{y}(k) = h(\mathbf{x}(k), \mathbf{v}(k)) \quad (2)$$

where $f : \Re^{n_\mathrm{x}} \times \Re^{n_\mathrm{u}} \times \Re^{n_\mathrm{w}} \rightarrow \Re^{n_\mathrm{x}}$ and $h : \Re^{n_\mathrm{x}} \times \Re^{n_\mathrm{v}} \rightarrow \Re^{n_\mathrm{y}}$ are nonlinear functions, $w_k \in \Re^{n_\mathrm{w}}$ and $v_k \in \Re^{n_\mathrm{v}}$ are i.i.d. random variables with pdf's p_w and p_v. The initial condition is known via $p_\mathrm{X}(0)$.

Let the state space be discretized (partitioned) into a finite number of sets called (state) cells, indexed by $s \in \mathcal{S} = \{1, 2, ..., S\}$. The index s is determined from

$$s = \arg\min_{s \in \mathcal{S}} \left\| \mathbf{x} - \mathbf{x}_s^{\mathrm{ref}} \right\|$$

where $\mathbf{x}_s^{\mathrm{ref}}$ are reference points (e.g., cell centers). Similarly, let the control action and measurement spaces be discretized into cells indexed by $a \in \mathcal{A} = \{1, 2, ..., A\}$ and $m \in \mathcal{M} = \{1, 2, ..., M\}$, respectively, and determined using reference points $\mathbf{u}_a^{\mathrm{ref}}$ and $\mathbf{y}_m^{\mathrm{ref}}$. The discretization results in $\mathcal{X} = \cup_{s=1}^{S} \mathcal{X}_s$, $\mathcal{U} = \cup_{a=1}^{A} \mathcal{U}_a$ and $\mathcal{Y} = \cup_{m=1}^{M} \mathcal{Y}_m$. Optionally, a 'sink cell', s_{sink} can be defined, where.the domain of a sink cell is the state space outside the domain of interest. A state is categorized as a sink cell, e.g., if $\min_{s \in \mathcal{S}} \left| x_i - x_{s,i}^{\mathrm{ref}} \right| > x_i^{\mathrm{lim}}$ for any state dimension i.

The evolution of the system can now be approximated as a controlled finite (state) Markov chain (CFMC) over the cell space. Using alternative terminology, in simple cell mapping (SCM) one trajectory is computed for each cell. Generalized cell mapping (GCM) considers multiple trajectories starting from within each cell, and can be interpreted in a probabilistic sense.

1.1 State evolution

Let the state pdf be approximated as a $S \times 1$ cell probability vector $\mathbf{p}_\mathrm{X}(k) = [p_{\mathrm{X},s}(k)]$ where $p_{\mathrm{X},s}(k)$ is the cell probability mass. The evolution of cell probability vectors is described by a Markov chain represented by a set of linear equations

$$\mathbf{p}_\mathrm{X}(k+1) = \mathbf{P}^{a(k)} \mathbf{p}_\mathrm{X}(k)$$

or, equivalently,

$$p_{\mathrm{X},s'}\left(k+1\right)=\sum_{s\in\mathcal{S}}p^{a}_{s',s}p_{\mathrm{X},s}\left(k\right)$$

where $\mathbf{P}^a$ is the transition probability matrix under action a, $\mathbf{P}^a=\left[p^{a}_{s',s}\right]$

$$p^{a}_{s',s}=\int_{\mathcal{X}_s}p\left(\mathbf{x}\left(k+1\right)\in\mathcal{X}_{s'}|\mathbf{x}\left(k\right)\in\mathcal{X}_s,u\left(k\right)\in\mathcal{U}_a\right)d\mathbf{x}.$$

1.2 State estimation

The likelihood of obtaining a measurement cell m, when the system state cell is s, is given by the likelihood matrix $\mathbf{L}$, $\mathbf{L}=\left[l_{m,s}\right]$

$$l_{m,s}=\int_{\mathcal{Y}_m}p\left(\mathbf{y}\in\mathcal{Y}_m|\mathbf{x}\in\mathcal{X}_s\right)d\mathbf{y}$$

Let a row in the likelihood matrix be denoted as a likelihood vector $\mathbf{l}_m$. Given the current likelihood vector and the previous posterior probability vector $\mathbf{p}_{\mathrm{X}}\left(k-1\right)$, a Bayesian estimate of the cell probability can be constructed,

$$\mathbf{p}_{\mathrm{X}}\left(k\right)\propto\mathbf{l}_m\otimes\mathbf{P}^{a(k)}\mathbf{p}_{\mathrm{X}}\left(k-1\right),$$

where $\otimes$ is the Haddamard product (component multiplication).

1.3 Controller design

1.3.1 Optimal control

In optimal control, the control task is to find an appropriate mapping (optimal policy or control table) π from states ($\mathbf{x}$) to control actions ($\mathbf{u}$), given the immediate costs $r\left(\mathbf{x}\left(k\right),\mathbf{u}\left(k\right)\right)$. The infinite-horizon discounted model attempts to minimize the geometrically discounted immediate costs

$$J\left(\mathbf{x}\right)=\sum_{k=0}^{\infty}\gamma^k r\left(\mathbf{x}\left(k\right),\pi\left(\mathbf{x}\left(k\right)\right)\right)$$

under initial conditions $\mathbf{x}\left(0\right)=\mathbf{x}$. The optimal control policy π^* is the one that minimizes J. The optimal cost-to-go is given by $J^*=\min_\pi J$.

Bellman's principle of optimality states that

$$J^*(\mathbf{x}) = \min_{\mathbf{u}} \left[r(\mathbf{x}, \mathbf{u}) + \gamma J^*(f(\mathbf{x}, \mathbf{u})) \right]$$

i.e., the optimal solution (value) for state $\mathbf{x}$ is the sum of immediate costs r and the optimal cost-to-go from the next state, $J^*(f(\mathbf{x}, \mathbf{u}))$. Application of the Bellman equation leads to methods of dynamic programming.

Value iteration In value iteration, the optimal value function is determined by a simple iterative algorithm derived directly from the Bellman equation. Let the immediate costs be given in matrix $\mathbf{R} = [\mathbf{r}^a]$, with column vectors $\mathbf{r}^a = [r_s^a]$, and collect the values of the cost-to-go at iteration i into a vector $\mathbf{J}^*(i) = [J_s^*(i)]$. Given arbitrary initial values $J_s^*(0)$, the costs are updated for $i = 0, 1, 2, ...$:

$$\begin{aligned} Q_s^a(i) &= r_s^a + \gamma \sum_{s' \in \mathcal{S}} p_{s',s}^a J_{s'}^*(i) \\ J_s^*(i+1) &= \min_{a \in \mathcal{A}} Q_s^a(i) \end{aligned}$$

$\forall s, a$, until the values of $J_s^*(i)$ converge. Denote the converged values by J_s^*. The optimal policy is then obtained from

$$\pi_s^* = \arg\min_{a \in \mathcal{A}} \left[r_s^a + \gamma \sum_{s' \in \mathcal{S}} p_{s',s}^a J_{s'}^* \right].$$

1.3.2 Predictive control

Given a system model and the associated costs, we can easily set up a predictive control type of a problem. In predictive control, the costs are minimized in an open loop in a fixed horizon

$$J(\mathbf{x}(k), ..., \mathbf{x}(k+H_\mathrm{p}), \mathbf{u}(k), ..., \mathbf{u}(k+H_\mathrm{p})) = \sum_{h=0}^{H_\mathrm{p}} r(\mathbf{x}(k+h), \mathbf{u}(k+h))$$

under initial conditions $\mathbf{x}$. In practice it is useful to introduce a control horizon, where it is assumed that the control action will remain fixed after a given number of steps, H_c. Often only one step is allowed and the optimization problem reduces to the minimization of

$$J\left(\mathbf{x}\left(k\right),...,\mathbf{x}\left(k+H_{\mathrm{p}}\right),\mathbf{u}\left(k\right)\right)=\sum_{h=0}^{H_{\mathrm{p}}}r\left(\mathbf{x}\left(k+h\right),\mathbf{u}\left(k\right)\right)$$

Exhaustive search Under control action a, the costs are given by

$$J_a=\sum_{h=0}^{H_{\mathrm{p}}}\left[\mathbf{r}^a\right]^T\mathbf{p}_{\mathrm{X}}\left(k+h\right)=\sum_{h=0}^{H_{\mathrm{p}}}\left[\mathbf{r}^a\right]^T\left[\mathbf{P}^a\right]^h\mathbf{p}_{\mathrm{X}}\left(k\right)$$

where $\mathbf{r}^a=\left[r_s^a\right]$ is a column vector of immediate costs and $\mathbf{p}_{\mathrm{X}}\left(k\right)$ is current state cell pdf. In order to solve the problem, it suffices to evaluate the costs for all $a\in\mathcal{A}$ and select the one minimizing the costs. The prediction horizon H_{p} is a useful tuning parameter; a long prediction horizon leads to mean level type of control.

The control policy mapping $\pi^{\diamond}$ can be obtained by solving the above problem in each state s and tabulating the results:

$$\pi_s^{\diamond}=\arg\min_a J_a.$$

Control horizons greater than one For many practical cases, a good controller design can be obtained using either the optimal control approach, or the predictive control approach with $H_{\mathrm{c}}=1$. Whereas the optimal control tends to result in "agressive" control actions in terms of the plant input (even if optimal in terms of the cost function), the predictive control approach provides a variety of responses as a function of the prediction horizon, H_{p}. With a small H_{p}, an agressive control is obtained. A large H_{p} results in mean level control where the closed loop shares the open loop plant dynamics. In some cases, however, an engineer may be interested in extending the controller design possibilities to larger control horizons. In principle, this is straightforward to realize in the CFMC context: One simply creates A different sequences of control actions, simulates the system accordingly, and selects the sequence that minimizes the cost function.

With large H_{c}, the search space of the exhaustive search can become too large for practical purposes, however. Luckily, in process engineering one is commonly interested in control sequences which fulfill rate constraints, one prefers to avoid jiggering, etc. With some simple rules, the set of appropriate control sequences can be reduced to a manageable size. Notice, however, that if sufficient information concerning the constraints is not contained in the cost description ($\mathbf{R}$), the optimal control policy (table) can not be constructed off-line. If the optimal control action is solved on-line, one is free to select the potential control sequencies based on any available information.

Example 1 Let us consider implementing a cost on $\Delta u(k)$, where the costs are described by $r(\mathbf{x}(k), u(k))$. If a predictive controller is to be designed off-line, the state $\mathbf{x}(k)$ must contain information about past control, $u(k-1)$, otherwise the function r does not have sufficient information for computing $\Delta u(k)$. If, on the contrary, the control action is designed on-line, one can omit control actions far from $u(k-1)$ from the set of control actions to be examined. This implements a hard limit on $\Delta u(k)$. Alternatively, one may change $\mathbf{R}$ at each sampling instant (on-line) so as to reflect its relation to past $u(k-1)$.

1.4 System analysis

The generalized cell-to-cell mapping is a powerful tool for analysis of nonlinear systems. In what follows, it is assumed that the system map (Markov chain) is described by transition probabilities $\mathbf{P}$. This may correspond to the process output under a fixed (open loop) control action a ($\mathbf{P} := \mathbf{P}^a$) or the systems closed loop behavior obtained from the construction of transition probabilities under $\mathbf{u} = \pi(\mathbf{x})$: $\mathbf{P}^\pi = \left[p^\pi_{s',s}\right]$, where

$$p^\pi_{s',s} = \int_{\mathcal{X}_s} p\left(\mathbf{x}(k+1) \in \mathcal{X}_{s'} | \mathbf{x}(k) \in \mathcal{X}_s, u(k) = \pi^*_s\right) d\mathbf{x}.$$

1.4.1 Simple cell mapping

Before proceeding to analysis using the generalized cell map (GCM), it can be enlightning to look at the case of simple cell mapping (SCM). A dominant SCM can be constructed from a GCM by selecting the most likely image cells and setting their probabilities to ones. Using the unravelling algorithm (Hsu, 1987, Sec 8.2.2), one can quickly examine the dynamic behavior of the dominant SCM (in hopes to grasp some of the essentials of the dynamics of the GCM). In particular, using the unravelling algorithm we obtain the persistent groups and their periods, as well as the step numbers for each of the cells to converge to one of these groups. Grouping cells with same group numbers gives the domain of attraction for a particular persistent group.

1.4.2 Recurrent, absorbing and transient cells

Recurrent cells can be located by studying the long term behavior of the finite Markov chains (FMC). The steady state probability distribution satisfies $\overline{\mathbf{p}}_\mathrm{X} = \mathbf{P}\overline{\mathbf{p}}_\mathrm{X}$ and, consequently, the distribution must be an eigenvector of $\mathbf{P}$. For the distribution to be a probability distribution, the eivenvalue must be one. Therefore, the recurrent cells are found by searching for the unit amplitude eigenvalues of $\mathbf{P}$; the nonzero elements of the associated eigenvectors $\overline{\mathbf{p}}_\mathrm{X}$ point to the recurrent cells. The cells that are not recurrent, are transient. Cells which have a zero probability for entering other cells, are absorbing.

Decomposing the probability vector into recurrent cells ($i_\mathrm{r} \in \mathcal{I}_\mathrm{r}$) and transient cells ($i_\mathrm{t} \in \mathcal{I}_\mathrm{t}$), the Markov chain can be written as follows:

$$\begin{bmatrix} \mathbf{p}_\mathrm{r}(k+1) \\ \mathbf{p}_\mathrm{t}(k+1) \end{bmatrix} = \begin{bmatrix} \mathbf{P}_\mathrm{rr} & \mathbf{P}_\mathrm{rt} \\ \mathbf{0} & \mathbf{P}_\mathrm{tt} \end{bmatrix} \begin{bmatrix} \mathbf{p}_\mathrm{r}(k) \\ \mathbf{p}_\mathrm{t}(k) \end{bmatrix}$$

As $k \to \infty$, the recurrent cells are visited infinitely often, whereas the transient cells are visited only finitely often. Among the recurrent cells, we can further classify the absorbing cells, ($i_\mathrm{a} \in \mathcal{I}_\mathrm{a}$):

$$\mathbf{P}_\mathrm{aa} = \mathbf{I}.$$

The absorbing states are never left, when visisted. The recurrent cells form communicating classes, where the cells within each communicating class communicate with each other, i.e., the probability of transition from one state to the other is nonzero. Each absorbing state only communicates with itself.

1.4.3 Stability, basin-of-attraction and absorbtion time

Analysis of stability and basin-of-attraction are based on the stationary transition probability map $\mathbf{P}^\infty = \lim_{k\to\infty} \mathbf{P}^k$. For periodic processes, a time average is used. The stationary distribution, starting from initial distribution $\mathbf{p}_0$, is obtained from $\mathbf{p}^\infty = \mathbf{P}^\infty \mathbf{p}_0$.

Stability The sink cell is an absorbing cell that represents the entire region outside the domain of interest. A nonzero probability to enter the sink cell indicates unstability of the system (given the resolution of the model).

Basin-of-attraction Examination of the behavior of transient cells as they enter the recurrent cells reveals the dynamics of the nonlinear system. We have that

$$\begin{aligned}\mathbf{p}_{\mathrm{r}}(k+1) &= \mathbf{P}_{\mathrm{rr}}\mathbf{p}_{\mathrm{r}}(k)+\mathbf{P}_{\mathrm{rt}}\mathbf{p}_{\mathrm{t}}(k)\\ &= \mathbf{P}_{\mathrm{rr}}\mathbf{p}_{\mathrm{r}}(k)+\mathbf{P}_{\mathrm{rt}}\mathbf{P}_{\mathrm{tt}}^{k}\mathbf{p}_{\mathrm{t}}(0)\end{aligned}$$

where $\mathbf{P}_{\mathrm{rt}}\mathbf{P}_{\mathrm{tt}}^{k}$ represents the conditional probability that a solution starting from a transient cell will pass into an recurrent cell at time $k+1$. The probability that this will eventually happen, $\mathbf{P}_{\mathrm{t2r}}$, is given by

$$\mathbf{P}_{\mathrm{t2r}}=\sum_{k=0}^{\infty}\mathbf{P}_{\mathrm{rt}}\mathbf{P}_{\mathrm{tt}}^{k}=\mathbf{P}_{\mathrm{rt}}\sum_{k=0}^{\infty}\mathbf{P}_{\mathrm{tt}}^{k}=\mathbf{P}_{\mathrm{rt}}\left(\mathbf{I}-\mathbf{P}_{\mathrm{tt}}\right)^{-1}$$

The basin-of-attraction of cell s' is given by the set of cell indexes s, where cells s have a nonzero probability of entering the cell s', i.e. $\arg p_{\mathrm{t2r},s',s}>0$. The size of the basin of attraction for a given recurrent state cell i is obtained by taking a sum of the elements in the row of $\mathbf{P}_{\mathrm{t2r}}$ associated with cell i, plus the stationary distribution associated with the cell i.

Absorbtion time The expected absorbtion time from the i'th state to the j'th state ($i\in\mathcal{I}_{\mathrm{t}}$, $j\in\mathcal{I}_{\mathrm{r}}$), $E\{\mathbf{k}\}$, is obtained from:

$$E\{\mathbf{k}\}=\mathbf{P}_{\mathrm{rt}}\sum_{k=0}^{\infty}k\mathbf{P}_{\mathrm{tt}}^{k}=\mathbf{P}_{\mathrm{rt}}\left(\mathbf{I}-\mathbf{P}_{\mathrm{tt}}\right)^{-2}.$$

1.5 Learning in GCM

Probabilistic transition maps are rarely available *a priori* for controlled processes. Therefore learning probability transition maps is a central task in CFMC's.

1.5.1 Construction of transition probabilities

If an accurate model (f, h) of the system is available, the CFMC transition matrix $\mathbf{P}$ can be constructed by sampling the model. An initial state vector, $\mathbf{x}_s$, is sampled uniformly from the support of cell s ($\mathbf{x}_s\in\mathcal{X}_s$), and the a'th control action, $\mathbf{u}_a^{\mathrm{ref}}$, is used as plant control input. The transition matrix is then constructed by simulating the plant model $\mathbf{x}_{s'}=f\left(\mathbf{x}_s,\mathbf{u}_a^{\mathrm{ref}}\right)$ for each

s and a, and counting the number of observed transitions from state s to s' ($n^a_{s',s}$):

$$p^a_{s',s} = \frac{n^a_{s',s}}{\sum_{s''=1}^{S} n^a_{s'',s}}$$

If the system state-action pairs are evaluated once ($e = 1$), a SCM map is obtained, i.e., the $\mathbf{P}$-matrix will be binary-valued. As the number of exhaustive evaluations e increases, a more and more refined CFMC map will be obtained. Note that there are multiple sources for 'uncertainty': discretization error (one initial state in a set $\mathcal{X}_s$ maps to state $\mathcal{X}_{s'}$, while another may map to $\mathcal{X}_{s''}$), noise $\{\mathbf{w}\}$ in states (in case of stochastic models), or noise in state measurements (not explicitely considered above), etc.

A similar Monte Carlo approach can be performed to compute the likelihood mass $l_{m,s}$ in the likelihood matrix $\mathbf{L}$. I.e., the system state is uniformly sampled from $\mathcal{X}_s$ and the plant output is constructed based on the measurement equation h. The likelihood matrix is then costructed from

$$l_{m,s} = \frac{n_{m,s}}{\sum_{m'}^{M} n_{m',s}}$$

1.5.2 Identification

If a model is not available, the CFMC transition probability matrices $\mathbf{P}^a$ can be learned from measured data, by counting the number of observed transitions. For one update, all that is required are two successive state vectors and a control: $\mathbf{x}(k), \mathbf{u}(k), \mathbf{x}(k+1)$, for a given sampling time T_s. Therefore, all past or on-line recorded data-triplets can be efficiently used. For likelihood matrix $\mathbf{L}$, a similar procedure can be applied.

Update of a large map may require an excessive number of samples, however, and parts of the model space may never be visited in real life, due to constraints in time and plant operation. This rises up the question of efficient and practical identification procedures. Typically, local updating procedures can be found and efficiently implemented. These are, however, always based on some a priori knowledge on the plant characteristics, such as smoothness or other structural information on f. Accompagned with careful design of experiments (statistics, interpolation / approximation techniques, focusing on control-relevant properties), the number of experiments required on real plant can be greatly reduced.

In real applications, the system state may not be measurable, or the measurement is severily corrupted by noise. If a system model ($\mathbf{P}^a$'s, $\mathbf{L}$) is available, a state estimator can be constructed. In some cases it can be more convenient to describe the system states based on delayed input–output measurements, leading to CMC–ARX models. If system inputs and outputs are measurable, the states of this type of models are always measurable. However, the state may not be minimal, and if the measurements are noisy a state estimator (an observer or a filter) may provide useful.

2 CFMC Toolbox

CFMC-toolbox is a MATLAB implementation of model conversion, controller design and open/closed loop analysis operations. First, a sample session illustrates the main functionalities of the CFMC-toolbox. This is followed by reference guide of the toolbox functions, in the next chapter.

2.1 Control design procedure

To start with, let us assume the following control design problem: We have a state-space model of the plant, and we have decided which input, state, and output variables to use. These tasks are far from trivial, but since the CFMC design can not help in this problem (albeit indirectly) we will assume that this selection has been made.

We now wish to design a controller that runs the plant output between given set points. A typical CFMC design procedure would involve the following (iterative) steps:

- Set **model resolution** by specifying discretization of plant inputs, states, outputs and output set points; and sampling time.
  - $\mathcal{X}$, $\mathcal{U}$, $\mathcal{Y}$, and $\mathcal{W}$; T_s in `Plant.Xref`, `Plant.Uref`, `Plant.Yref` and `Plant.Wref`; `Plant.Ts`.
- Set control **targets** by specifying immediate costs:
  - $r(\mathbf{x}, \mathbf{u})$ in `Cost.R`.
- Build a **GCM plant model** (by successive evaluations of the original model, and counting the occurred state transitions).
  - $\mathbf{P}^a$'s and $\mathbf{L}$ in `Plant.f(a).P` and `Plant.h.L`
  - Implemented in `gcm_generate_model`.
- Analyze the **behavior** of the GCM plant model.
  - simulate step responses, i.e. response from a given initial state using a fixed control action (implemented in `gcm_step`)

  - analyze plant map from plant input to plant output. Implemented in:
    - `gcm_eiganalysis`, (examine absorbing/ recurrent/ transient cells, probabilities of transition, expected times)
    - `gcm_pinf` (examine stationary distributions)
    - `gcmbofa` (examine basins-of-attractions for recurrent states, i.e. stationary analysis)
    - `gcm_comm` (examine communicating classes)

- Design an optimal or predictive **controller**, based on the CFMC plant model presentation:
  - solve for π^* in `Cost(q).Pi`, where `q` referes to a set point
  - Implemented in `gcmoptipol` or `gcmpredpol`.
  - Examine different controller parameter settings (implemented in `gcmccp`).

- Build a **GCM closed-loop model** of the closed-loop, formed by the original model and the controller.
  - build the closed-loop map
  - implemented in `gcmclosed`.

- Analyze the **behavior** of the GCM closed-loop model:
  - Simulate performance of designed controller using original plant model. The reference and disturbances are defined in structure `Sim`. (Implemented in `gcmsim`)
  - Simulate set point control, i.e., a step in set point, from a given initial state (implemented in `gcm_free`)
  - analyze closed-loop map (from setpoint to plant output) using:
    - `gcm_eiganalysis`
    - `gcm_pinf` (compute stationary map)
    - `gcm_comm` (sort cells into communicating classes)

2.2 Computational load and memory requirements

The steps involved in model building and controller design may be time and memory intensive. The following list outlines the essential components affecting computational load, time and memory requirements:

- The $S \times n_{\mathrm{x}}$ state reference points need to be stored in memory, as well as $S \times A$ immediate costs. Assuming a grid-discretization, the number of cells will be $S = \prod n_i$, where n_i is the resolution (number of discretizations) in the i'th dimension of the state.
- When building a GCM plant model, in general, all $S \times A$ discrete states need to be visited. To have a 1%-unit precision for the distribution of probability transfer, at least 100 evaluations of the original model are required, for each state-action pair. Consequently, $100SA$ evaluations would be required. The $S \times S \times A$ elements of the probability transition matrices need to be stored, as well as $S \times A$ counter values.
- In optimal control, the Bellman equation needs to be solved, where the matrix $\mathbf{Q}$ is of size $S \times A$. In solving predictive control problems, cumulative cost matrices of size $A \times S$ need to be stored. The time required to solve predictive problems is directly related with the horizon length, H_{p}.

It clear that the memory requirements are far from feasible. The remedy, however, is in that the probability transition matrices are *sparse*. Noticing that most of the elements in the probability transition matrices are zeros, the memory requirements (as well as computational load) are greatly reduced. With the help of MATLAB sparse matrix facilities, the computations can be easily implemented using sparse matrices. Another central requirement in MATLAB coding is the use of *vectorization* wherever possible.

2.3 Example session

In what follows, the main features needed for initializing and running the CFMC toolbox are covered.

Example 2 (LTI-DSS) The script `fun_test.m` provides an example of the code functionalities. Typing `fun_test` in the MATLAB command window shows a sample example on how to build a CFMC model, design a optimal controller, and how to analyse the system. The plant to be controlled is a linear time-invariant (LTI) discrete time system.

Example 3 (nonlinear ode) If the original plant model is based on an ode-description, the continuous time plant equations need to be solved Ts times ahead. These can efficiently be solved using ode23 routines, and exploiting the batch for for solving the equations. The script `vdv5_test` provides an example on predictive multivariable control based on plant ode-description of a nonlinear CSTR (continuous stirred tank reactor) system governed by van der Vusse (vdv) equations.

2.3.1 System equations (`Plant` fields `funf`, `funh`, `Ts`, `Xname`, `Uname`, `Yname`)

The system model is given in a discrete-time state space form in two m-files provided by the user. They provide means for computing DSS-state and measurement equations. Names of these files are stored as strings in the `Plant`-structure, in fields `funf` and `funh`.

The basic syntax for the system equations (1)–(2) is

```
Xk1  =  funf(Xk,Uk,<Ts>)
 Yk  =  funh(Xk)
```

where the state, control and measurement are in the respective columns of the variables; `Ts` is the sampling time (scalar) stored in a respective field in the `Plant` structure.

Names of the state, control and output variables are stored in the `Plant` structure, in respective fields: `Xname`, `Uname` and `Yname`.

Remarks

- The functions `funf` and `funh` should be implemented so that input state can be given as a matrix (with N columns). The output arguments should return the system state and measurement for each of the N columns.
- If the original system model is an ode (i.e., continuous time), the `funf` should perform the sampling (i.e., solve the ode `Ts` time units ahead from the initial state).
- The functions are used for simulation purposes, and may include noise sources.

- It is convenient to code initial / nominal info in (or via) `funf` and `funh` (see sample files for syntax). This includes information on the default sampling time, as well as nominal (typical) values, min–max range, and names of the variables (state, control and measurement). These functionalities can be used by automatic CFMC model set-up procedures.

Example 4 (LTI-DSS) The system DSS equations are implemented in files `funf.m` and `funh.m`. The plant set up procedure is illustrated in `fun_plant_setup.m`. which reads the initial/nominal information from `funf` and `funh`.

Example 5 (nonlinear ode) The CSTR system DSS equations are used via `vdvf5.m` and `vdvh.m`. These files accept as their input argument a column matrix of initial states. The system equation M-function (`vdvf5.m`) runs a vectorized ode-solver (`gcm_vodesolver`), which then uses the ode described in `vdvdv5.m` . The parameters of the CSTR are specified in a separate M-script, `vdvparam`.

The `gcm_vodesolver` takes as its input argument a handle for the system ode-equations, a column matrix of system states, the control input vector, and the sampling time. It returns a column-matrix of states at the next sampling instant. The task of the vectorized ode-solver is to solve the equations for the (presumable large amount of) initial states in an efficient batch-wise manner. In order to do so, it uses Matlab's standard ode-solvers.

Therefore, the system equation ode need to be given in a standard odefile in format `xdot=odefile(t,x,u)`. However, the odefile must accept the system equations to be solved simultaneously for multiple state vectors. Here's the trick: The multiple input vectors are given in the input argument `x`, which is constructed by catting state vectors on top of each other. Consequently, the output argument `xdot` must also be a vector of same size as `x`. In order for the above scheme to work, the odefile must understand that the 'pseudo'-state `x` in fact contains multiple state vectors.

In general, the system equations remain more transparent if each row represents a state, and the different state vectors are handled by each column. In the remaining code, each column of `x` can then be handled as if it was a separate system state. The function `gcm_v2v` performs the necessary vector-to-matrix and matrix-to-vector transitions.

This somewhat cumbersome procedure enables efficient vectorized solving of the ode-equations, yet preserving the transparency in implementation of the ode's. Recall that spending too much time for solving the ode equations soon makes the CFMC approach unfeasible (!)

The measurement equation m-file (`vdvh.m`) simply returns the second state as the output measurement.

2.3.2 System discretization (`Plant` fields `Xref`, `Uref`, `Yref`, `Wref`)

A fundamental step in CFMC is discretization of the state, control and measurement spaces. Information on these is stored in the `Plant`-structure, in fields `Xref`, `Uref`, `Yref` and `Wref`.

State and measurement spaces The discretization of state space is based on reference points. Each point in the continuous space is mapped to a cell associated with the closest reference point. The discretization consists of `S` reference points, given as column vectors in `Xref` of size $n_{\mathrm{X}} \times S$. The (output) measurement space is discretized in a similar manner, in `Yref`, $n_{\mathrm{Y}} \times M$.

- In what follows, a grid-based implementation is considered. Any set of reference points can be used, but the computational complexity of evaluating the mapping from a multidimensional point in real space to an index in a discrete space (implemented in m-file `gcm_x2s`) soon makes other approaches infeasible. In a grid-based approach, the mapping will require unidimensional searches only.
- The existance of a sink cell is indicated by the presence of a reference state with infinite components. A sink cell in the output measurement spaceis defined in a similar fashion. Any point in the real space which is outside of the region half the distance between two outermost discretizations, in any grid direction, will be considered to map into the sink cell.

Action and reference spaces The action space is the set of possible control inputs. Each is a column vector given in `Uref`, which consists of `A` actions of size $n_{\mathrm{U}} \times A$. The reference set `Wref` consists of a set of Q possible targets (set points in optimal control), $n_{\mathrm{Y}} \times Q$. It can, but need not, be the same as the measurement space.

Example 6 (LTI-DSS) A sample file `fun_plant_setup` illustrates how to generate a grid-based discretization. Discretization is given for each dimension separately, and `gcm_inigrid` is used to create a multidimensional hypercube. The main m-script `fun_test` calls the `fun_plant_setup` m-file. After generating the discretizations, a vector with infinite elements is added to both state and measurement reference matrices, `Xref` and `Yref`, indicating the use of a sink cell.

Example 7 (nonlinear ode) The discretization of the five-state van der Vusse CSTR is given in `vdv5_plant_setup`.

2.3.3 Model conversion (`Plant` fields `f` and `h`, with fields `P`, `L` and `Nsum`)

Given the discretization and a plant model, a CFMC (GCM) map can be constructed. The procedure consists of simulating the system equations from a random initial point within each state cell. The simulations are conducted for each control action. The resulting image states are observed. A counter of transitions from each state cell–action pair to the next cell is updated and stored in the memory. Repeating this a large number of times allows to build a probability density map: the transition probability distribution.

The command for building a CFMC model (converting the model in system equation -form to CFMC form) is as follows

```
Plant = gcm_generate_model(Plant)
```

This command updates the fields `f` and `h` in the `Plant`-structure.

Structure of `Plant.f` and `Plant.h` `f` is a vector where each element contains information on a particular control action. `f` itself is a vector structure with fields `P` (probability transition maps) and `Nsum` (count of transitions from a cell). The element $p^a_{s',s}$ of the probability transition map, $\mathbf{P}$, is obtained from

$$\mathtt{Plant.f(a).P(sdot, s)}$$

Similarly, `Plant.f(a).P(:,s)` gives the probability distribution (in a frequentist sence) of transitions from cell `s` under action `a`. `Plant.f(a).Nsum(s)` gives the number of observed transitions from cell `s`, under action `a`.

The field `h` contains similar information for the output measurement: in fields `Nsum` (counts) and `L` (likelihoods). Hence

$$\mathtt{Plant.h.L(m, s)}$$

gives the likelihood of observing measurement m when in state s, `Plant.h.L(:,s)` gives the probability distribution for the output measurement (again, in a frequentist sence), where `s` is the current state-cell.

Remarks

- Fields `P`, `L` and `N` are stored as sparse matrices. Using MATLAB-command `full` converts from sparse to full.

- When `gcm_generate_model` is used for the first time (when there's no field `f` in `Plant`), the initial state for each cell will be the reference point `Plant.Xref`. For consequtive evaluations, a uniformly distributed random point within the hypercube is used as an initial point. Typically, tens of realizations should be averaged in order to get a feasible probability transition map.

- Use `gcm_step` to simulate trajectories of probability densities from a given initial point, under a fixed control action.

Example 8 (LTI-DSS) The example `fun_test` calls repeatedly for `gcm_generate_model`. From time to time, a controller is redesigned and the closed-loop performance is assessed, so as to determine when a sufficient number of plant samples has been simulated.
We find the state corresponding to nominal state using `gcm_x2s`:

$$\begin{aligned} &\texttt{[Ts,Xnom,Unom]=funf;} \\ \texttt{snom} &= \texttt{gcm\_x2s}(\texttt{Xnom},\texttt{Plant.Xref}); \\ \texttt{anom} &= \texttt{gcm\_x2s}(\texttt{Unom},\texttt{Plant.Uref}); \end{aligned}$$

Eigenvalue analysis provides us information on absorbing, recurrent and transient states, as well as largest expected times of transitions:

$$[\texttt{EA},\texttt{P\_t2r},\texttt{Et\_t2r},\texttt{Et\_stat}] = \texttt{gcm\_eiganalysis}(\texttt{Plant},\texttt{anom});$$

In `EA`, the fields `ipa`, `ipr` and `ipt` list cells that are absorbing, recurrent or transient. `EA.ipa` always includes the sink cell. `Et_stat` is a structure with a field `maxEt`, which gives the maximum expected time of transition to each of the recurrent cells. A wealth of other information is also provided. The relationships between the recurrent cells can be analysed using

$$\texttt{CClass} = \texttt{gcm\_comm}(\texttt{Plant},\texttt{anom});$$

where CClass is an array where each element lists indexes to states that form a communicating class. A sink cell will always form one communicating class with one member only.

The step response trajectories of the transition probabilities from a given state under a given control action are simulated using `gcm_step`. A plot of the step reseponse from the nominal state under the nominal action is obtained using:

$$\texttt{gcm\_step}(\texttt{Plant},\texttt{snom},\texttt{anom})$$

The size of basin-of-attraction can be examined using `gcmbofa`. It computes the size of basin of attraction for each output state, for each step input (in steady state). Sizes of basins-of-attraction can be further projected to a subset of all states. If the first state is the plant output, then projecting towards the first state reveals size of the basin of attraction for each plant output:

$$\texttt{BofA} = \texttt{gcmbofa}(\texttt{Plant});$$
$$\texttt{gcmbofa}(\texttt{BofA},[],1,\texttt{anom});$$

Example 9 (nonlinear ode) The sample script for updating the CSTR model/controller is in `vdv5_test`.

2.3.4 Control design

Two types of control design are supported: optimal and predictive. In both control designs, the control spesifications are given via immediate costs (costs at each sampling instant). The overall cost, minimized in the procedure, consists of minimization of the sum of future immediate costs, either in an discounted infinite horizon, or a finite horizon. Assuming that the plant state is unique, certain and known, both solutions can be presented as a deterministic control policy. This policy can be tabulated as a function of state cell s (and set point q). Control design consists then of

- a) setting the immediate costs,
- b) setting the parameters adjusting the future horizon, and
- c) solving the optimization problem.

Control specifications (`Cost` fields `R` and `controllertype`, and `gam` or `H_p`) The `Cost` structure contains the immediate costs for being in cell s and taking action a, given a target reference q:

$$\texttt{Cost}(\texttt{q}).\texttt{R}(\texttt{s},\texttt{a})$$

Example 10 (LTI-DSS) For the example case, costs are defined in `fun_control_setup`. The m-file sets field `R` (immediate costs) of the `Cost` structure using squared error at the output. A high cost is set for the sink cell.

Example 11 (nonlinear ode) The same costs (squared deviation on plant output) in `fun_control_setup` can be used in the CSTR-case, too.

Two types of controllers can be defined, specified in field `controllertype`:

- An optimal controller uses value iteration to solve for the optimal (set point) control policy. `controllertype` is set to `'optimal'`. The discount factor is given as an additional parameter `Cost(q).gam`. The controller is solved using `gcmoptipol`.
- A predictive controller uses value exhaustive search to solve for the optimal control policy. `controllertype` is set to `'predictive'`. The prediction horizon is given as an additiona parameter `Cost(q).H_p`. The controller is solved using `gcmpredpol`.
- For the time being, only control horizons of 1 are considered. M-file `gcmpredpoldp` implements `H_c` larger than one (not fully integrated to toolbox, yet).

Solving for controllers (`Cost` fields `Pi` and `Jstar`) The numerical solutions of the controllers are obtained by running either of the following commands for each set point q: For optimal controller:

$$\texttt{Cost} = \texttt{gcmoptipol}(\texttt{Plant}, \texttt{Cost})$$

For predictive controller

$$\texttt{Cost} = \texttt{gcmpredpol}(\texttt{Plant}, \texttt{Cost})$$

The optimal control policies (tables) are stored in `Cost(q).Pi(s)`, which gives the optimal control action if the system is in cell s and the target (cell) is q.

Remarks

- An approximate relation between γ and H_p is given by, $H_\mathrm{p} \approx 1/(1-\gamma)$.
- `gcmoptipol` and `gcmprepol` can also be solved for one set point only, by using q as a third argument.
- The solution routine for the optimal controller stores the optimal cost-to-go in the `Cost`-structure field `Jstar`. For predictive control, the expected cost in the prediction horizon is stored.

2.3.5 System analysis (`Sim` fields `WW` and `x0,y0,p_x0, ...`)

A closed loop CFMC map is constructed using

$$\texttt{Plant\_cl} = \texttt{gcmclosed}(\texttt{Plant}, \texttt{Cost})$$

The resulting CFMC map is a description of the closed loop system, consisting of the plant (in `Plant`) and controller (`Cost`). The `Plant_cl`-structure is a copy of the `Plant`-structure, with the following exceptions:

$$\texttt{Plant\_cl.f}(\texttt{q}).\texttt{P}(\texttt{s}, \texttt{sdot})$$

will describe the probability of evolution of the closed loop system from cell s to cell s with the q'th set point controller. Also `Uref` is replaced by `Wref`, as well as `Uname` by `Wname`, and `Wname` is set to '`closed-loop`'.

Stability The stability of closed-loop mapping can be examined by looking at the probabilities of entering the sink cell. The percentage of stable ($p = 0$), unstable ($p = 1$) and potentially unstable ($0 < p < 1$) cells are computed and plotted for each set point q using

$$\texttt{gcmstability}(\texttt{Plant\_cl})$$

Communicating classes and speed of response The cells of the closed-loop mapping can be analyzed using `gcm_comm`:

$$\texttt{CClass} = \texttt{gcm\_comm}(\texttt{Plant\_cl})$$

which returns the list of indexes to cells that form communicating classes. An overview to the associated points in real space can be obtained using `gcm_dispref`, e.g.

$$\texttt{gcm\_dispref(CClass\{1\},Plant.Xref)}$$

A more detailed analysis of properties of communicating classes can be obtained by using the probability transition matrix as the input argument. For example, for a given set point q:

$$\begin{aligned} \texttt{P\_cl} &= \texttt{Plant\_cl.f(q).P;} \\ [\texttt{CClass},\texttt{BofA\_cells},\texttt{Et\_cells}] &= \texttt{gcm\_comm(P\_cl)} \end{aligned}$$

returns also the cells that map to a particular communicating class with a non-zero probability (`BofA_cells`). These cells form the basin-of-attraction of this communicating class. The expected time of absorbtion to the communicating class is returned as the third output argument (`Et_cells`).

Example 12 (LTI-DSS) The fastest and slowest transition to the first communicating class for the 1'st set point is found and plotted using the following commands. First let us generate information on the communicating classes

$$\begin{aligned} \mathtt{q} &= \mathtt{1} \\ \mathtt{P\_cl} &= \mathtt{Plant\_cl.f(q).P;} \\ [\mathtt{CClass},\mathtt{BofA\_cells},\mathtt{Et\_cells}] &= \mathtt{gcm\_comm(P\_cl)} \end{aligned}$$

Then we look for indexes to cells from where the fastest and slowest trajectories depart:

$$\begin{aligned} \mathtt{ia} &= \mathtt{find(Et\_cells\{1\}} == \min\mathtt{(Et\_cells\{1\}))} \\ \mathtt{sa} &= \mathtt{Bofa\_cells\{1\}(ia); sa = sa(1);} \\ \mathtt{ib} &= \mathtt{find(Et\_cells\{1\}} == \max\mathtt{(Et\_cells\{1\}))} \\ \mathtt{sb} &= \mathtt{Bofa\_cells\{1\}(ib)} \end{aligned}$$

Finally, we plot these trajectories

$$\begin{gathered} \mathtt{T = 100;} \\ \mathtt{figure(1); gcm\_step(Plant\_cl, sa, q, T);} \\ \mathtt{figure(2); gcm\_step(Plant\_cl, sb, q, T);} \end{gathered}$$

Simulation Just as with the plant model, the closed-loop step response (set point response) of the state pdf can be simulated and plotted using `gcm_step`

$$\texttt{gcm\_step}(\texttt{Plant\_cl}, \texttt{s}, \texttt{w})$$

where s is the current system cell, and w is the reference cell index. Closed-loop set point responses can also be plotted using the `gcm_free` and `gcm_freek` commands:

$$\texttt{gcm\_free}(\texttt{Plant}, \texttt{Cost}, \texttt{s}, \texttt{q}, \texttt{T});$$
$$\texttt{gcm\_freek}(\texttt{Plant}, \texttt{Cost}, \texttt{s}, \texttt{q}, \texttt{T});$$

The former computes the trajectory using the CFMC model and plots the response in terms of state pdf. The latter computes the trajectory using the DSS model (`funf`), and shows a (sample) trajectory in real space.

Given the designed controller, "true" plant model with initial conditions, and a reference trajectory, the closed loop system can then be simulated, using `gcm_sim`. Simple plots of the plant behavior are produced in terms of the evolution of states, outputs and system control actions for the specified reference trajectory. The `Sim`-structure contains following fields: `WW` contains the target trajectory, desired outputs for each value of k are given in respective rows. Fields `x0`, `y0` and `p_x0` are column vectors that define the initial state for simulation.

Example 13 (LTI-DSS) An example of a simulation set-up is given in script `fun_simulation_setup`. It generates a random step/ramp sequence based on nominal info on the system.

Example 14 (nonlinear ode) The `fun_simulation_setup` can also be used for initializing the `Sim`-structure in the CSTR-case.

3 Reference guide

This section provides a detailed function reference with examples. The M-files can be categorized into two sets of M-functions (or M-scripts): application dependent and general purpose functions.

3.1 Application dependent files

The toolbox comes with a set of descriptions for two example cases, the fun-set and the vdv5-set. These file and structure names are examples provided with the toolbox; in new applications these are to be replaced by descriptions corresponding to the application in question. Notice, that file and structure names can be changed, but names for fields in structures must be followed exactly.

3.1.1 Linear transfer function (LTI-DSS)

`fun_test` is a start-up script that runs the CFMC toolbox example for controlling a linear system.

- `fun_plant_setup` is a sample model set-up.
- `fun_control_setup` is a sample control set-up.
- `fun_simulation_setup` is a sample simulation set-up.
- `funf`, `funh` is a discrete-time state-space DSS plant description for a linear second order non-minimum phase system

  $$\frac{Y(s)}{U(s)} = k\frac{\beta s + 1}{(\tau_1 s + 1)(\tau_2 s + 1)}$$

  with $k = 2$; $\beta = -5$, $\tau_1 = 2$ and $\tau_2 = 10$.
- `funf1`, is a discrete-time state-space DSS plant description for a first order system $\frac{Y(s)}{U(s)} = \frac{k}{\tau s+1}$, where $k = 2$, $\tau = 5$.

3.1.2 van der Vusse CSTR (nonlinear ode)

`vdv5_test` is a start-up script that runs the CFMC toolbox example for controlling a nonlinear system.

- `vdv5_plant_setup` is a sample model set-up.
- `vdvf5`, `vdvh` is a sampled van der Vusse plant description. Van der Vusse plant is a CSTR (continuous stirred tank reactor) which is has non-monotone havior and non-linear non-minimum phase dynamics. The original model is in ode-form, in `vdvdv5` and `vdvparam`.

3.2 Structures

The information on plant and controller is collected in structures. The `Plant` and `Cost` structures are described in the following. Some of the analysis data are also collected in structures, for details on these see function references for `gcmsim` (`Sim`-structure), `gcmbofa` (`BofA`-structure), `gcm_comm` (`CClass`-structure), and `gcm_eiganalys` (`EA`-structure).

`Plant-structure` The `Plant`-structure contains information on discretization, the CFMC model, and sampling time, among other things. Discretization information is presented as column vectors of reference points for plant states, inputs, outputs and controller set points (control targets). In general, points in continuous spaces map to the closest reference point in respective discrete space. Also names for these variables are stored, to be used in plots and alike. The `Plant`-structure stores the CFMC-model of the plant, i.e., the transition probability matrices and counters of observed transitions. The `Plant`-structure contains the following fields:

- `f`-vector structure of size A, with fields
  - `P`, a sparse $S \times S$ transition probability matrix. The transition probability from cell s to cell s' (*sdot*) under action a is stored in

    `Plant.f(a).P(sdot,s)`

  - `Nsum`, a S-vector of counters of observed transitions from cell s. The number of observed visits to cell s (under action a) is obtained from `Plant.f(a).Nsum(s)`.
- `h`. a structure with fields

    - L, a sparse $M \times S$ likelihood probability matrix. The likelihood of observing a measurement m when in cell s is stored in

      $$\texttt{Plant.h.L}(\texttt{m},\texttt{s})$$

    - Nsum, a S-vector of counters of observed measurements, when in cell s.

- Ts, sampling time (scalar)
- Uname, an array containing string(s) for name(s) of the control variables. Uname{d} contains the string for the d'th variable.
- Uref, a $n_{\mathrm{U}} \times A$ matrix of column vectors of plant control inputs. Plant.Uref(:,a) specifies the a'th control action.
- Wname, an array containing string(s) for name(s) of the target variables.
- Wref, a $n_{\mathrm{Y}} \times Q$ matrix of column vectors of controller set points. Plant.Wref(:,q) specifies the q'th set point.
- Xname, an array containing string(s) for name(s) of the state variables.
- Xref, a $n_{\mathrm{X}} \times S$ matrix of column vectors of state cell referenc. The reference state associated with the s'th (state) cell is specified in

  $$\texttt{Plant.Xref}(:,\texttt{s})$$

- Yname, an array containing string(s) for name(s) of the output variables.
- Yref, a $n_{\mathrm{Y}} \times M$ matrix of column vectors of state cell reference points. Plant.Yref(:,m) specifies the reference measurement associated with the m'th output cell.

Cost-structure The Cost-structure contains information on control specifications, controller parameters, and control policy. Control specifications are given as immediate costs for each state cell-action pair. Controller parameters depend on controller type (optimal, predictive). Control policy is a table of control action to apply in each state cell. The Cost-structure is a vector structure, where Cost(q) contains information on the q'th set point controller. Each controller is specified by the following fields:

- `gam`, a scalar discount factor (in optimal control)
- `H_c`, a scalar control horizon (in predictive control with `gcmpredpoldp`)
- `H_p`, a scalar prediction horizon (in predictive control)
- `Jstar`, a S-vector of expected discounted costs(optimal control), or expected costs in the prediction horizon (predictive control).
- `Pi`, the control policy. For the q'th set point controller, the control action a to apply when in state cell s is obtained from `Cost(q).Pi(s)`. The real-valued control **u** to be applied to the plant is obtained using

$$\begin{aligned} \mathtt{a} &= \mathtt{Cost(q).Pi(s)} \\ \mathtt{u} &= \mathtt{Plant.Uref(:,a)} \end{aligned}$$

- `R`, a $S \times A$ matrix of immediate costs. For the q'th set point controller, with its set point at `Plant.Yref(:,q)`, the immediate costs for being in state s and applying an action a are obtained from `Cost(q).R(s,a)`.

3.3 Function reference

3.3.1 fun*

Sample files for the fun-example, see Section 3.1.

3.3.2 gcm_allpnts_int

Create indexes to all elements in a hypercube where each dimesion contains a given number of elements

$$\mathtt{COORD} = \mathtt{gcm\_allpnts\_int(HC)}$$

- `HC` is a m-vector of the number of elements in each dimension
- `COORD` a list of all different combinations of vector elements, `prod(HC)`$\times m$. Each row lists the indexes to one combination.

This function is used for creating points in a grid of a given size. It is implemented as recurrent function.

Example 15 Indexes to all elements in a two-dimensional square, with two elements in one direction and three elements in the second direction are obtained as follows:

```
gcm_allpnts_int([2 3])
```

which returns a 6x2 matrix

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 2 \\ 2 & 3 \end{bmatrix}$$

Uses: gcm_allpnts_int

3.3.3 gcm_automeq

Take a random element `e` from a list `v`

```
e = gcm_automeq(v)
```

where each element of the vector `v` has an equal probability to be selected.

Example 16 To select randomly one element from list [1,4,17,22] use the following command: `gcm_automeq([1 4 17 22])`, returns, e.g., 22.

3.3.4 gcmbofa

Analyse basins-of-attraction. The command

```
BofA = gcmbofa(Plant, Cost)
```

computes a vector structure `BofA` and plots the results. Its `i`'th element `BofA(i)` has the following fields

- `BOFA(q,:)` is a vector of the sizes of the basins-of-attraction for set-point `q`, projected to the `i`'th dimension of the state.
- `S(q)` is the sum of sizes of basins-of-attractions
- `In` is the discretization of the i'th dimension.

The function works as follows: The function first computes a closed-loop `Plant`-structure (using `gcmclosed`). An existing closed-loop structure can be used with the syntax $\texttt{BofA} = \texttt{gcmbofa}(\texttt{Plant\_cl})$. It then generates lists of projections and setpoints to consider (default: all dimensions and all setpoints). The set of dimensions to consider can be limited by listing them as a third argument: $\texttt{BofA} = \texttt{gcmbofa}(\texttt{Plant}, \texttt{Cost}, \texttt{ixlist}, \texttt{iqlist})$, where `ixlist` is a vector of indexes to desired dimensions. The set of setpoints to consider can be limited in a similar manner, by specifying it as a fourth argument, `iqlist`. Each setpoint is then handled separately by computing eigenanalysis information for associated the closed-loop map (using `gcm_eiganalysis`) and computing sizes of basins-of-attraction (`gcm_bofa_size`) for the desired projection dimensions. Results are stored in the `BofA`-structure.

A plot of the results is produced by using the `BofA`-vector structure as the first argument:

$$\texttt{gcmbofa}(\texttt{BofA})$$

The plot consists of a number of subplots, one for each projected state dimension. Each subplot shows a set of bars indicating the sizes of basins-of-attraction in percentage (y-axis) for a value of the state (x-axis). Different colors for the bars indicate different setpoints. For a state associated with the controlled output one would expect a peak at the desired value, for an uncontrolled state a flat distribution might be expected. The commands for plotting are implemented as a subfunction within `gcmbofa` (`gcm_bofa_plot`).

Example 17 To plot the steady-state distribution of the second state for all set points under CFMC control, use the following commands:

```
load vdvdata10
BofA=gcmbofa(Plant,Cost);
gcmbofa(BofA,[],2);
```

This results in the plot in figure 1

Uses: gcm_bofa_size, gcmclosed

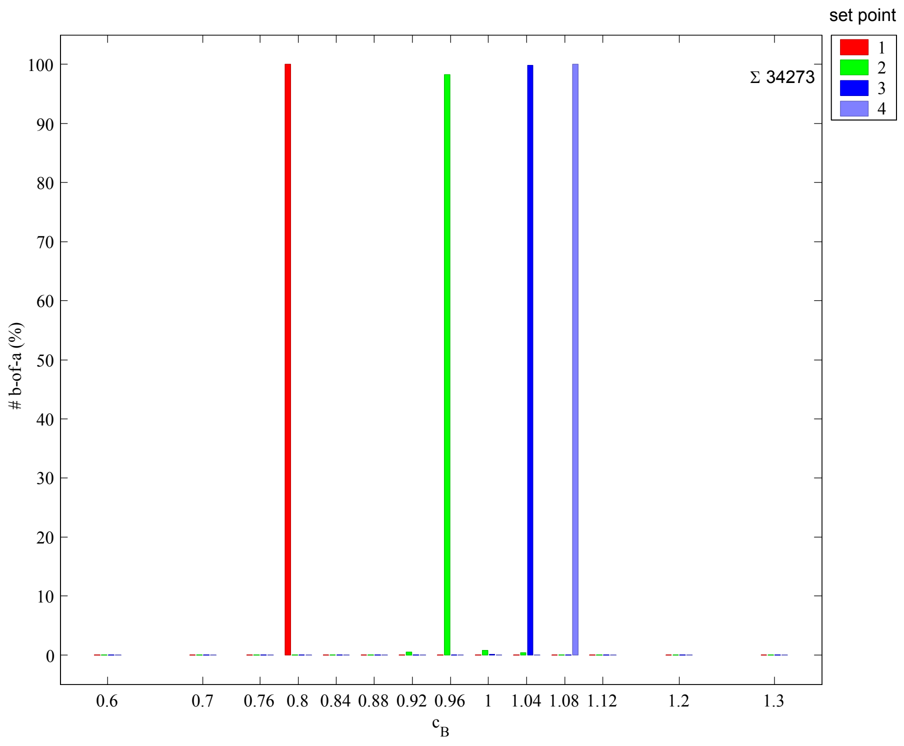


Figure 1: gcmbofa

3.3.5 gcm_bofa_size

Estimate the sizes of basins-of-attraction for the recurrent cells of system described by transition probability matrix `P`:

$$\texttt{BofA\_size} = \texttt{gcm\_bofa\_size}(\texttt{P})$$

- `BofA_size` is a `lipr`- row vector containing estimated sizes of basins-of-attraction for each recurrent cell (`lipr` is the number of recurrent cells).

The function first looks for recurrent and transient cells (using `gcm_eiganalysis`), computes the stationary mapping (using `gcm_pinf`), and finally computes the sum of stationary probabilities for entering a particular recurrent state. With additional input arguments `BofA_size = gcm_bofa_size(P,EA,P_t2r)`, results from a previos eigenanalysis can be used.

The sizes of basins-of-attraction can be further projected towards a given state dimension using

`[BofA_size,P_rr_inf,BofA_proj,In] = gcm_bofa_size(P,Xref_proj)`

where `Xref_proj` is an S-dimensional row vector of values towards which the results are projected, i.e., information on the s'th cell is cumulatively projected to value `Xref_proj(s)`. `BofA_proj` returns a sum of sizes of basins-of-attraction for all cells with the same value. `In` is a list of different values in `Xref_proj`. With no output arguments, a bar plot is produced (`In` vs. `BofA_proj`). With additional input arguments, results from a provious eigenanalysis can be used: `gcm_bofa_size(P,EA,P_t2r,Xref_proj)`.

Example 18 Let that the second state be associated with the plant output ($y = x_2$ is the controlled variable) and the reference states be given in `Plant.Xref`. The sizes of the basins-of-attraction of the plant outputs can be obtained as follows:

```
load vdvdata10    % load Plant and Cost structures
Plant_cl = gcmclosed(Plant,Cost)    % create closed-loop maps
P2 = Plant_cl.f(2).P;    % closed-loop FMC for q=2

% solve eigenvalues and static map from transient to recurrent
cells:
[EA,P_t2r]=gcm_eiganalysis(P2);

% sizes of b-of-a, projected towards output:
[BofA_size,P_rr_inf,BofA_proj] = gcm_bofa_size(P, EA, P_t2r,
Plant.Xref(2,:));
```

The `BofA_size`-vector will have 152 elements (as many as there are recurrent cells in the system, `EA.ipr`). For example, `BofA_size(26)` is 396.46, which means that the sum of probability cell masses of initial cells which converge to cell nro 26 is about 396. This is about 1.2% of the total cell mass ($S = 34273$). The value is a real number, since it is obtained by summing up the transition probabilities, and a trajectory from a given cell may converge to several final cells (with some probability). For recurrent cells within a communicating class, the static distribution is used.

The projected sizes of the b-of-a are obtained from `BofA_proj`. It contains 14 elements as there are 14 different values in `Plant.Xref(2,:)`. For example, `BofA_proj(8)`is 33682.08 which means that more than 98% of the total cell mass converges to this projected value. This corresponding value of the output is obtained from `In(8)`, and is 0.96. This is the closest discretized value of x_2 for set point $q = 2$, $w = 0.95$.

The plot contains raw information of the plot produced by `gcmbofa` (unscaled, only for set point 2):

```
gcm_bofa_size(P, EA, P_t2r, Plant.Xref(2,:));    % without output
args, a plot is produced
```

Uses: gcm_pinf, gcm_get_partitions, gcm_eiganalysis

3.3.6 gcmccp

Change controller parameters:

```
Cost  =  gcmccp(Cost,gamma)
Cost  =  gcmccp(Cost,H_p)
```

Changes the `Cost`-structure so that an optimal controller with infinite discounted horizon with decay-parameter `gamma`, or so that a predictive controller with control horizon `H_p`, respectively, is defined for all set point controllers.

With an additional input argument, a list of indexes to set point controllers can be provided, for which the changes are to appear.

Example 19 The `vdvdata10` contains an optimal controller, designed using $\lambda = 0.98$ (discount factor):

```
load vdvdata10
gcmdisp(Cost);
```

Let us change the discount factor to 0.95 and recompute the controller for the 1'st set point:

```
Cost = gcmccp(Cost,0.95)
Cost = gcmoptipol(Plant,Cost,1);
```

Let us then examine the predictive control design, by setting the prediction horizon to 5:

```
Cost = gcmccp(Cost,5)
Cost = gcmpredpol(Plant,Cost,1);
```

Uses: gcmdisp

3.3.7 gcm_chbase

Change base to and from a 10-number system

$$\begin{aligned} \texttt{idx} &= \texttt{gcm\_chbase}(\texttt{v}, \texttt{vmax}) \\ \texttt{v} &= \texttt{gcm\_chbase}(\texttt{idx}, \texttt{vmax}) \end{aligned}$$

where

- `vmax` is a row vector that specifies the number of elements in each basis direction
- `v` is a matrix where each row vector specifies a vector in the basis vmax
- `idx` is a matrix where each row vector specifies a vector in the basis of 10 (i.e., corresponds to `vmax = [10 10 10 ...]`

Indexes are counted from 1 upwards.

Example 20 Let the basis be given by `vmax = [3 3 2]` (e.g., a $3 \times 3 \times 2$ dimensional cube). Then `gcm_chbase(1,vmax)` gives [1 1 1], `gcm_chbase(2,vmax)` gives [1 1 2], `gcm_chbase(3,vmax)` gives [1 2 1], ..., `gcm_chbase(17,vmax)` gives [3 3 1], and `gcm_chbase(18,vmax)` gives [3 3 2]. Upper limit is not implemented, so `gcm_chbase(19,vmax)` will give [4 1 1].

The inverse mapping is obtained when length of `v` and `vmax` is the same, e.g., `gcm_chbase([3 3 1],vmax)` gives 17

3.3.8 gcm_check_for_grid

Check if the input matrix seems to be built using a regular grid. This function performs a couple of tests to see if it contains all the elements that would exist in a discretized hypercube. Syntax:

$$[\texttt{YN}, \texttt{In}] = \texttt{gcm\_check\_for\_grid}(\texttt{Xref})$$

returns

- `YN=1` if tests are passed ok (if `Xref` seems to be from a grid), or `YN=0` if not.
- `In` is an array containing different values that are found for each of the dimensions.

This function is vital, e.g., in searching closest match of an input vector and reference vectors, as one-dimensional searches are sufficient for a regular grid. In current implementation, two tests are performed:

- Let n_i denote the number of different elements in the i'th dimension (`n(i)=length(In{i})`). If product of n_i equals the number of columns in `Xref`, then test 1 is passed.
- If the matrix `Xref` does not contain two similar column vectors next to each other, then test 2 is passed.

It is possible that a matrix passes the above tests, and yet is not built using points from a regular grid (more tests should/can be implemented). If the `Xref` contains infinite elements, its last column is omitted from the checks.

Example 21 For the vdv-example, the following commands

```
load vdvdata10
[YN,In]=gcm_check_for_grid(Plant.Xref);
```

show that the data comes from a grid (YN is 1). The discretization for, say, the third dimension is obtained using In{3}, which contains a vector [90 100 105 110 115 125].

Uses: gcm_get_partitions

3.3.9 gcmcla

Run a set of closed-loop analysis tools:

```
[Sink,BOFA,Sim]=gcmcla(Plant,Cost,Sim)
```

This collection of analysis tools does the following:

- builds a closed-loop CFMC for all set point controllers (using `gcmclosed`)
- examines stability of the closed-loop systems (using `gcmstability`, see for `Sink`)
- examines sizes of basins-of-attraction (using `gcmbofa`, see for `BOFA`)
- runs a test simulation (using `gcmsim`, see for `Sim`)
- plots general info (using `gcmdisp`)

Uses: gcmclosed. gcnstability, gcmbofa, gcmsim, gcmdisp

3.3.10 gcmclosed

Generate a CFMC model for a closed loop system:

```
Plant_cl = gcm_closed(Plant,Cost)
```

where `Plant` and `Cost` specify the plant and controllers to be used, `Plant_cl` contains the closed-loop model. In particular, the policy in `Cost(q).Pi(s)` is used at each state cell s. `Plant_cl.f(q).P` contains the closed-loop FMC model for the q'th set point.

The `Plant_cl` will be a copy of the `Plant`-structure with exeptions in the replacement of the following fields:

- `Uref` and `Uname` are copied from `Wref` and `Wname` (in `Plant`)
- `Wname` is set to 'closed-loop system'
- for all `q` and `s`, `f(q).P(:,s)` is replaced by `Plant.f(a).P(:,s)` with `a=Cost(q).Pi(s)`. Hence, a closed-loop map is obtained where. the open loop system in `Plant` is controlled by the policy in `Cost`:

  $$c^q_{s',s} := p^{\pi^q(s)}_{s',s}$$

  where $c^q_{s',s}$ is the transition probability from cell s to s' of the closed-loop system under set point controller q; π^q is the control policy for set point q, and $p^a_{s',s}$ is the open-loop plant transition probability from cell s to s' under action a. The closed-loop CFMC-model will contain the transition probabilities c.

- for all `q` and `s`, `f(q).Nsum(s)` is replaced by `Plant.f(a),Nsum(s)` with `a=Cost(q).Pi(s)`

If an additional input argument `q` is given, the function returns the CFMC map for the q'th set point only.

Example 22 A closed-loop map for all set points is generated as follows:

```
load vdvdata10
Plant_cl = gcmclosed(Plant,Cost);
```

The structure `Plant_cl` contains four closed-loop FMC models, one for each set point. For the q'th set point, the probability transition matrix for the controlled system is obtained from `Plant_cl.f(q).P`.

3.3.11 gcm_comm

Look for communicating classes among the cells in a system described by the probability transition matrix:

$$\texttt{CClass} = \texttt{gcm\_comm(P)}$$

where `P` is a probability transition matrix.

- `CClass` is an array of C vectors, where C is the number of communicating classes: Each vector in the array contains a list of indexes to recurrent cells that communicate with each other. The communicating classes are ordered such that the largest classes appear first, and smallest classes last.

Alternatively a `Plant`-matrix is can be used as an input argument, from where the P-matrix corresponding to a map under the `a`'th control action is used in further analysis. If the `Plant`-matrix corresponds to a closed-loop system, then the `q`'th set point is considered:

$$\begin{aligned}\texttt{CClass} &= \texttt{gcm\_comm(Plant,a)}\\ \texttt{CClass} &= \texttt{gcm\_comm(Plant\_cl,q)}\end{aligned}$$

With additional output arguments, the basins of attraction associated with the communicating classes and the expected times of entering the class are extracted:

```
[CClass,BofA_cells,Et_cells] = gcm_comm(P,plim)
```

where

- `BofA_cells` is an array of C vectors. The c'th vector contains indexes to cells that map to the c'th class with probability larger than `plim`. (default `plim = 0.001`).
- `Et_cells` is an array of C vectors. The c'th vector contains expected times of entering the c'th class. The expected time in `Et_cells{c}{i}` corresponds to the expected time from the initial cell `BofA_cells{c}(i)` to the `c`'th communicating class.

The gcm_comm works as follows. First, eigenanalysis is performed (using `gcm_eiganalysis`), in order to find the recurrent cells. The algorithm starts from the first recurrent cell, looks for cells which have a nonzero probability of entering this cell, and adds them to a list. It then checks the cells that have a nonzero probability of entering any of the cells in the list, and updates the list until no new cells are found. All the cells in the list then form a communicating class. The algorithm then proceeds in a similar fashion with the remaining recurrent cells, until all cells are examined. Classes are then ordered such that the largest class is the first, etc.

If additional output arguments are given, the algorithm looks for the elements in the basin-of-attraction of each communicating class. Class by class, the algorithm examines the static map from transient to recurrent cells (from `gcm_eiganalysis`), and looks for cells with nonzero probability of entering the particular class. The basin-of-attraction consists of these transient cells plus the recurrent cells contained in the particular class. The expected times of transition are computed using expected times of transition from transient to recurrent cells (from `gcm_eiganalysis`), and weighing them with the probability of entering a particular recurrent cell (recall that a trajectory from a transient cell may have a nonzero probability of converging to several communicating classes).

Example 23 Define $\mathbf{P}$ as follows:

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

i.e., the first cell is a recurrent (absorbing) cell, the second cell is a transient cell which maps to the first in one step; the third and fourth cells are recurrent cells which map to each others in one step; the fifth cell maps to the second in one step.

Applying `gcm_comm(P)`

```
[CClass,BofA_cells,Et_cells]=gcm_comm(P)
```

the following results are obtained: The first communicating class contains cells 3 and 4 (`CClass{1}`); the second communicating class contains the first cell (`CClass{2}`). The basin-of-attraction to the first communicating class contains cells 3 and 4 (`BofA_cells{1}`); the b-of-a for the second class contains cells 1, 2 and 5 (`BofA_cells{2}`). The expected times of transition are 0 for both cells in the first class (`Et_cells{1}`), 0, 1.0 and 2.0 for cells in the second class (`BofA_cells{2}`), as the first cell is already in the communicating class, the second cell maps to the communicating class in one step, the fifth cell maps to the first in two steps - via the second cell.

Uses: gcm_dispref, gcm_eiganalysis

3.3.12 gcmdisp

Display info on plant discretization to command window

```
gcmdisp(Plant)
gcmdisp(Cost)
```

- With `Plant`-structure as input argument, information on original model M-file names, sampling time, discretization and number of plant model evaluations is provided.
- With `Cost`-structure as input argument information on size of the system, control design method and parameters, as well as ranges of immediate costs and expected costs is provided.

Example 24 `load vdvdata10; gcmdisp(Plant)` gives the following info on the plant model:

```
== MODEL INFO =============================================
- state equations:  'vdvf5'
- measurement equations:  'vdvh'
== DISCRETIZATION =========================================
Sampling time:  0.00555556
References w:  Q=4
w(1):  {0.8, 0.95, 1.05, 1.09} (4)
Outputs y:  M=15
y(1):  {0.6, 0.7, ..., 1.2, 1.3} (14)
States x (grid):  S = 34273 incl.  sink cell
x(1):  [0.85, 3.55], {1, 1.3, ..., 3.1, 3.4} (17)
x(2):  [0.55, 1.35], {0.6, 0.7, ..., 1.2, 1.3} (14)
x(3):  [85, 130], {90, 100, ..., 115, 125} (6)
x(4):  [85, 130], {90, 100, ..., 115, 125} (6)
x(5):  [97.5, 117.5], {100, 105, 110, 115} (4)
Controls u:  A=60
u(1):  {3, 6, ..., 30, 35} (12)
u(2):  {-9000, -3000, -1113.5, -800, 0} (5)
===========================================================
N=10 evaluations on all state-action pairs.
===========================================================
```

For each variable, the discretized values are shown as {x$_1$,x$_2$,...,x$_n$} (n), where n is the number of discretizations. If the state cell reference contains a sink cell, the domain inside region of interest is shown as [x$_{\min}$,x$_{\max}$].

`gcmdisp(Cost)` gives the following info on the controller_

```
==== CONTROLLER INFO ==================================
A DSS control system with:
34273 states
60 actions
4 set points
optimal control design, gamma=0.98 (H=50)
======================================================
Immediate costs in R: min mean max
q = 1:  0.00000 < 0.19171 < 10.00000
q = 2:  0.01000 < 0.15743 < 10.00000
q = 3:  0.01000 < 0.17886 < 10.00000
q = 4:  0.01000 < 0.19743 < 10.00000
```

```
======================================================
Expected costs in J*:  min mean max
q = 1:  0.00000 < 0.81711 < 499.99584
q = 2:  0.49998 < 0.99485 < 499.97637
q = 3:  0.49998 < 1.22111 < 499.97637
q = 4:  0.49997 < 1.40221 < 499.97278
======================================================
```

The H is an approximate 'equivalent prediction horizon', $H = \frac{1}{1-\lambda}$.

Uses: gcm_check_for_grid, gcm_get_partitions

3.3.13 gcm_dispref

Display values from a set of reference points. The command:

```
gcm_dispref(Xref)
```

gives textual info on the values in `Xref`. `Xref` is a matrix of column vectors, where each vector contains a reference point. For each dimension of the vectors, a list of different values is produced, the most frequent being marked by an asterisk. The full command

```
Xml=gcm_dispref(ss,Xref)
```

enables to specify separately the columns `ss` of `Xref` to consider, i.e., the same as `gcm_dispref(Xref(:,ss))`. If the output argument `Xml` is given, no textual output is produced, but the function returns a single reference point (the 'most frequent' one).

Example 25 `load vdvdata10; gcm_dispref(Plant.Xref(:,1:6))` displays the following info:

```
A set of 6 states:
x(1) = { 1* }
x(2) = { 0.6* }
x(3) = { 90* }
x(4) = { 90*, 100 }
x(5) = { 100*, 105*, 110 , 115 }
```

which indicates that all size column vectors have 1 as their first element, 0.6 as their second element, and 90 as their third element. Fourth element is either

90 (most often) or 100. The fifth element is either 100 or 105 (most often), 110 or 115.

A most likely element in the set is obtained by calling the function with an output argument: `Xml=gcm_dispref(Plant.Xref(:,1:6))`. The obtained `Xml` is [1 0.6 90 90 100].

Uses: gcm_get_partitions

3.3.14 gcm_eig

Find indexes to recurrent, transient and absorbing states of a system described by a sparse probability transition matrix `P`, using eigenvector decomposition:

$$[\texttt{ipr}, \texttt{ipt}, \texttt{ipa}] = \texttt{gcm\_eig}(\texttt{P}, \texttt{K}, \texttt{Sigma})$$

where `ipr`, `ipt` and `ipa` are lists of indexes to recurrent, transient and absorbing states, respectively.

`gcm_eig` uses Matlab's `eigs` to perform the computations, except for very small matrices. By default, the following parameters are used: $\texttt{K} = 100$, $\texttt{Sigma} = 1 + \varepsilon$ (see `eigs`). The resulting eigenvectors and eigenvalues are rounded to the third digit. Recurrent cells are found by looking for eigenvectors of length 1, when the associated nonzero eigenvalues point to recurrent cells. Absorbing cells are found by looking for unity probabilities at the main diagonal of `P`. They are also included in the recurrent cell index list.

Example 26 Define $\mathbf{P}$ as follows (see example on `gcm_comm`):

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The command `[ipr,ipt,ipa]=gcm_eig(P)` results in $\texttt{ipr} = [1\ 3\ 4]^\mathsf{T}$, $\texttt{ipt} = [2\ 5]^\mathsf{T}$ and $\texttt{ipa} = 1$, i.e., cells 1, 3 and 4 are recurrent and cells 2 and 5 are transient. Cell 1 is also an absorbing cell.

3.3.15 gcm_eiganalysis

Compute eigenvalues and perform basic analysis of a system specified by a probability transition matrix transition. The basic syntax is

$$\texttt{EA} = \texttt{gcm\_eiganalalysis(P)}$$

where `P` is a probability transition matrix. `EA` is a structure with the following fields:

- `ipa` are indexes to absorbing states $p_{i,i} = 1$, if $i \in$ `ipa`
- `ipr` are indexes to recurrent states (including absorbing states). These cells are entered infinitely often.
- `ipt` are indexes to transient states. These cells are entered only finitely often.

Alternatively a `Plant`-matrix can be used as an input argument, from where the P-matrix corresponding to a map under the `a`'th control action is used in further analysis. If the `Plant`-matrix corresponds to a closed-loop system, then the `q`'th set point is considered:

$$\begin{aligned}\texttt{EA} &= \texttt{gcm\_eiganalalysis(Plant,a)}\\ \texttt{EA} &= \texttt{gcm\_eiganalalysis(Plant\_cl,q)}\end{aligned}$$

First, the eigenvectors and eigenvalues of `P` are computed and cells are classified into absorbing, recurrent and transient based on eigenanalysis (using `gcm_eig`). Since `gcm_eig` uses the Matlab's function for sparse matrices `eigs` (with parameter `K` for the number of eigenvalues to find), the computations are repeated for the cells judged as transient, as long as no new recurrent states appear (i.e., at least twice). One pass of `gcm_eig/eigs` can be forced by setting `PRECISION_FLAG` to zero: `gcm_eiganalalysis(P,[],K,PRECISION_FLAG)`.

If a second output argument is specified, the stationary transition probability matrix from transient to recurrent states is computed: `[EA,P_t2r] = gcm_eiganalalysis(P)`. This can be a computing power and memory consuming task. The code functions as follows: the $\mathbf{P}$-matrix is decomposed into $\mathbf{P}_{\mathrm{rr}}$ and $\mathbf{P}_{\mathrm{rt}}$, and the direct inversion is attempted: $(\mathbf{I} - \mathbf{P}_{\mathrm{tt}} + \varepsilon\mathbf{I})^{-1}$, where ε is a very small number. If this fails (which is common with large $\mathbf{P}$) a second strategy is adopted. Recall that $\mathbf{P}_{\mathrm{t2r}} = \mathbf{P}_{\mathrm{rt}} (\mathbf{I} - \mathbf{P}_{\mathrm{tt}}) = \mathbf{P}_{\mathrm{rt}} \sum_{k=0}^{\infty} \mathbf{P}_{\mathrm{tt}}^{k}$, so the inverse can be approximated recursively. This is slow, but monitoring the memory consumed by of the sparse cumulative product $\mathbf{P}_{\text{tt}}^k$ and removing small elements (using `gcm_squeezep`) makes it possible to compute an approximation of the inverse. The iterations terminate when the approximation error is less than 1%, or more than 200 transition steps have been computed.

The expected transition time is computed for a third output argument, as well as some handy statistics on transition times (average `meanEt` and worst case `maxEt`) returned in the fourth structure:

$$[\texttt{EA}, \texttt{P\_t2r}, \texttt{Et\_t2r}, \texttt{Et\_stat}] = \texttt{gcm\_eiganalalysis(P)}$$

Example 27 Consider the following **P** matrix (see example on `gcm_comm`):

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Running the command $[\texttt{EA}, \texttt{P\_t2r}, \texttt{Et\_t2r}, \texttt{Et\_stat}] = \texttt{gcm\_eiganalalysis(P)}$ displays following info on the P-matrix analysed, the analysis results, and on computation of inverse:

```
GCM_EIGANALYSIS...
Computing eigenvectors & eigenvalues of P of size 5x5..  ok.
1 absorbing / 3 recurrent states / 2 transient states
Computing probabilities of transition to recurrent states...
P_t2r:  Computing inverse...  ok.
...  ( 0) ok.
```

The following results are returned in the output arguments:
· `EA` contains fields `ipa` (1), `ipr` ($[1\ 3\ 4]^\mathsf{T}$) and `ipt` ($[2\ 5]^\mathsf{T}$) indicating absorbing, recurrent and transient cells.
· `P_t2r` is a sparse matrix

$$P_{\mathsf{t2r}} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

indicating that the two cells in the list of transient cells (2 and 5) both map to the first cell in the list of recurrent cells (cell 1), with probability 1.

· `Et_t2r` is a sparse matrix structured in the same way as `P_t2r`, indicating the expected times of transition

$$E\left\{t_{\mathsf{t2r}}\right\} = \begin{bmatrix} 1 & 2 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

· `Et_stat` gives information on the transition time from transient to recurrent cells: mean 1.5 and maximum 2 for the first cell.

Uses: gcm_eig, gcm_squeezep

3.3.16 gcm_estimate_x

Construct an estimate of the plant cell distribution. The command

```
[s,xdk,p_x_est,x_e] = gcm_estimate_x([],Plant,ymk,a,p_x)
```

first estimates the index m for the discretized plant output `Plant.Yref(:,m)` (using `gcm_estimate_y`), and then computes a Bayesian estimate for the system cell distribution

$$\mathbf{p}_{\mathrm{X}}\left(k\right) \propto \mathbf{l}_m \otimes \mathbf{P}^{a(k)}\mathbf{p}_{\mathrm{X}}\left(k-1\right)$$

- `p_x_est` refers to $\mathbf{p}_{\mathrm{X}}\left(k\right)$, computed by

$$\mathtt{Plant.h.L(m,:)'.*(Plant.f(a).P*p\_x)}$$

- `x_e` is the average of reference points (weighted by corresponding cell mass probabilities). ML or median state can be obtained by suitable flagging in the M-file.
- `s` is the index to the closest matching reference cell, i.e., `s = gcm_x2s(x_mean, Plant.Xref)`
- `xdk` is the corresponding discrete reference point, `xdk = Plant.Xref(:,s)`.
- `ymk` is the output measurement
- `a` is the control action $a\left(k\right)$

- `p_x` (input argument) corresponds to the past cell distribution ($\mathbf{p}_X(k-1)$).

If the Bayesian inference fails (measurement is 'not possible'), the measurement is trusted and a warning is produced.

When evoked using only the first two input arguments:

$$\texttt{s = gcm\_estimate\_x}(\texttt{xmk}, \texttt{Plant})$$

a measured state vector `xmk` (the state measurement) is assumed to be available. The cell index is `s` obtained by looking for the closest reference cell (using `gcm_x2s`).

Example 28 Following examples illustrate the command when the full state is measured:

```
load vdvdata10
s = gcm_estimate_x( Plant.Xref(:,1114),Plant)
```
results in $s = 1114$.

```
[s,xdk,p_x_est]=gcm_estimate_x( 1.01*Plant.Xref(:,1114),Plant)
```
results in $s = 1114$, `xdk = Plant.Xref(:,14)` and `p_x_est` a sparse matrix with 1114'th element equal to 1, zero elsewhere.

If only the plant output is measured, a ML estimate is constructed. Let the current state cell estimate be 1114, the control action a be 1, and the measured output be 0.95 and 0.94:

```
e=sparse(34273); e(1114)=1;
[s1,xdk1,p_x_est1]=gcm_estimate_x( [],Plant,0.95,1,e)
[s2,xdk2,p_x_est2]=gcm_estimate_x( [],Plant,0.94,1,e)
```

In first case the estimated state cell is 1114, in the second it is 946.

Uses: gcm_x2s, gcm_estimate_y

3.3.17 gcm_estimate_y

Construct an estimate for the plant output. The command

```
[m,ydk]=gcm_estimate_y(ymk,Plant)
```

finds the index `m` to the closest match among output reference points. `ydk` is the associated output vector in the reference set, `ydk = Plant.Yref(:,m)`.

A ML based estimate of the plant output is constructed using

```
[m,ydk]=gcm_estimate_y(ymk,Plant,p_x)
```

where `p_x` is the state cell mass distribution, and the output distribution is obtained from `p_y` = `Plant.h.L*p_x`.

Uses: gcm_x2s

3.3.18 gcm_eucl_norm

Compute Euclidean norm between two matrices. For two matrices given in variables `U` and `V`, consisting of column vectors of size $I \times S_1$ and $I \times S_2$, the Euclidean norm is computed by

```
E=gcm_eucl_norm(U,V,W)
```

where `W` is a diagonal weighing matrix ($\mathbf{W} = \mathbf{I}$ by default). `en` is a $S_1 \times S_2$ matrix:

$$E_{s,r} = \sqrt{\sum_{i=1}^{I} \left[W_{i,i} \left(\mathbf{U}_{i,s} - \mathbf{V}_{i,r}\right)\right]^2}$$

This function is used, e.g., to generate immediate costs based on deviation between set point and output vectors.

Example 29 Define matrices **U** and **V** as follows:

$$\mathbf{U} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{V} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

The following commands compute Euclidean norms between two vectors, a vector and a three column vectors, and between all combinations of the three column vectors

`gcm_eucl_norm(U(:,1),V(:,2))`
returns $E = 2.24$.

`E=gcm_eucl_norm(U(:,1),V)`
returns a vector:

$$\mathbf{E} = \begin{bmatrix} 0 & 2.24 & 3.16 \end{bmatrix}$$

E=gcm_eucl_norm(U,V)
returns a matrix:

$$\mathbf{E} = \begin{bmatrix} 0 & 2.24 & 3.16 \\ 1.41 & 1 & 3.16 \\ 1.41 & 2.24 & 2 \end{bmatrix}$$

3.3.19 gcm_ff

Update figure plot. `gcm_ff` is a shorthand command for `figure(gcf); drawnow`

3.3.20 gcm_free

Simulate and plot a discrete (closed-loop) systems' free trajectory from a given initial state:

```
gcm_free(Plant_cl,s0,q)
gcm_free(Plant,Cost,s0,q)
```

where the closed loop system is defined either by the CFMC model `Plant_cl`, or the plant `Plant` and controller `Cost`. The initial state is given in `s0` as the cell index s (a scalar integer). Alternatively a probability distribution vector (a S-dimensional vector) can be specified as the initial state. q is the index to the set point controller to be considered.

An additional argument `T` can be given to control the time scale of the simulation. If `T` is a scalar, it is the end-time of the simulation (in real time, vs. sampling time given in the `Plant_cl` or `Plant`-structure). If `T` contains two elements, the first is the start time, the second is the end-time of simulation. If `T` is a vector, the corresponding points in time are returned (sampling time should be the same as in the model).

A further additional argument `ilist` can be given to specify the states for which the trajectory is to be plotted (default is all states).

The function can also be used for generating open-loop step responses. However, for step responses it is more straightforward to use `gcm_step` directly.

Example 30 Plots of free responses of the closed-loop system (for $q = 3$) from a given initial state ($s = 1114$) are obtained as follows:

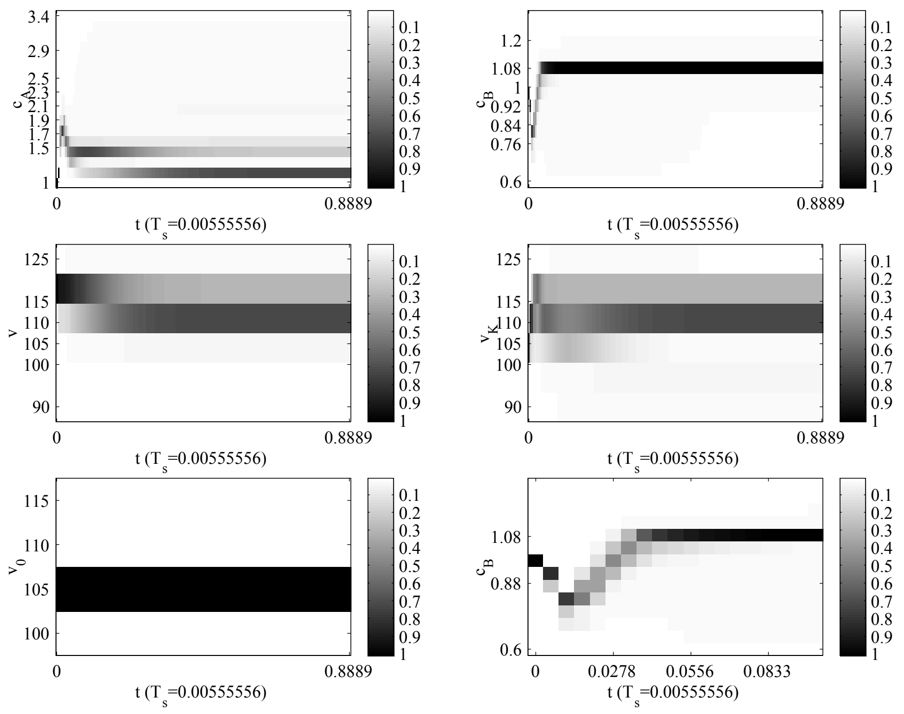


Figure 2: gcmfree

```
load vdvdata10
gcm_free(Plant,Cost,1114,3);

subplot(3,2,6);
gcm_free(Plant,Cost,1114,3,[0 0.1],2);
colormap(gray);
```

The resulting plot is shown in Figure 2. The plots illustrate the trajectories of the probability distributions for each state. The colors indicate the probability mass (colorbar shown at the left of each figure). The time scale is obtained from sampling time in `Plant.Ts` (0.0056 sec). The bottom right plot is a 'zoom' of the plant output from time 0 to 0.1. By default, the colormap is set to `colorcube`.

Uses gcm_step (gcm_simp) to generate the state trajectories.

3.3.21 gcm_freek

Simulate and plot the original (closed-loop) systems' free trajectory from a given initial state:

```
gcm_freek(Plant_cl,s0,q)
gcm_freek(Plant,Cost,s0,q)
```

The closed loop system is defined either by the CFMC model `Plant_cl`, or the plant `Plant` and controller `Cost`. The initial cell is given in `s0` as the cell index (a scalar integer). This means that the state `Plant.Xref(:,s0)` is used as the initial state in simulation. Alternatively, any state vector of appropriate size can be given as the input argument. The controller for the q'th set point is used in simulations.

With the latter form, gcm_freek has the same syntax as `gcm_free`. Instead of `Plant.P`, it uses the original model specified in `Plant.funf` to simulate the plant. If a previously unobserved transition is observed, a warning is produced.

If an output argument is given, the state trajectories are returned and no plot is produced. A second output argument is returns a `Sim`-structure for an equivalent simulation (to be used by `gcmsim`).

The function can also be used for generating open-loop step responses using the original plant model (in `Plant.funf`):

```
gcm_freek(Plant,s0,a)
```

Example 31 The resulting plot is similar to that from `gcm_free`, but in real state space:

```
load vdvdata10
gcm_freek(Plant,Cost,1114,3)
gcm_freek(Plant,Cost,1114,3,[0 0.1],2);
```

results in a plot of Figure 3.

Uses: gcm_check_for grid

3.3.22 gcm_generate_model

Generate a CFMC map by sampling an original plant DSS model:

```
Plant = gcm_generate_model(Plant)
```

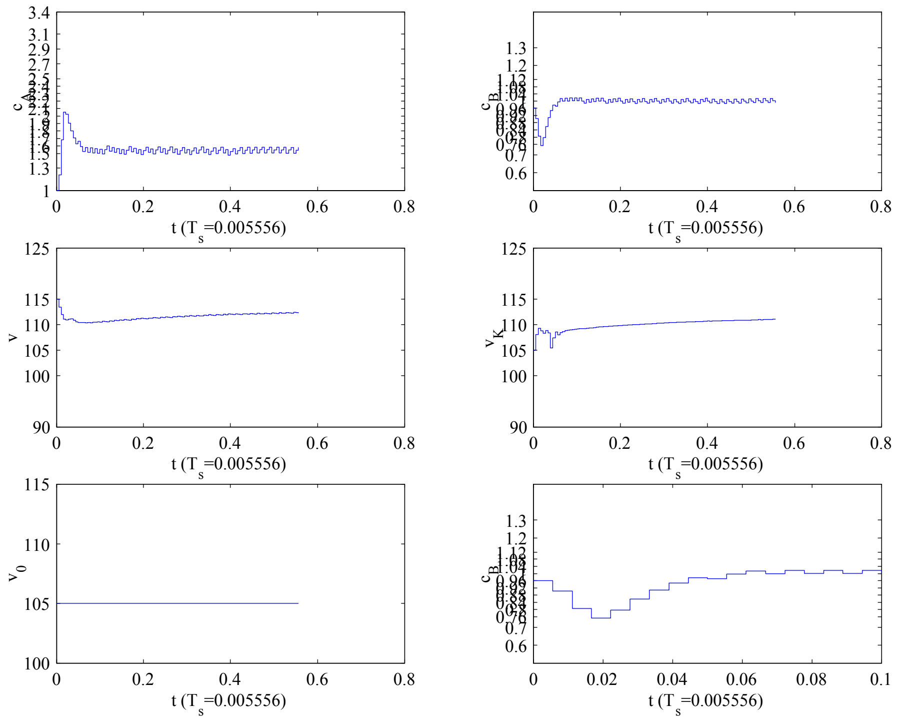


Figure 3: gcmfreek

The `Plant` structure must have the following information:

- discretization information in fields `Xref`,`Yref`,`Uref`
- information of names of original DSS model in fields `funf`, `funh`
- plant sampling time `Ts`

The following fields in `Plant` structure will be updated :

- `f`, a vector structure with fields `P` and `Nsum`
- `h`, a structure with fields `L` and `Nsum`

The function works as follows: First, information on `Plant`-model structure is displayed. If the model is evaluated for the first time, probability transition matrices etc. are initialized (see code line 40). The code then checks a) if a sink cell exists (if the set `Xref` contains any infinite elements, it is assumed that the S'th cell is a sink cell), b) if the points in `Xref` seem to make a grid (a grid is very important from efficiency point of view, non-grid systems are doable but slow). For each control action, the following steps are performed:

- Initial states are generated. For the first evaluation, `Xref`-points are used. For consequtive evaluations, initial points are generated from a uniform distribution within the support of each cell. See `gcm_sampleu`.
- State equations are solved, starting from initial states. For efficiency it is assumed that the `funf` and `funh` accept multiple initial values, i.e., the syntax for `funf` is `XX1 = feval(funf,XX,U,Ts)` where `XX` and `XX1` are matrices consisting of initial and one-step ahead states in column vectors; the control action `U` is a vector (same for all initial states).
- Simulation results are examined to find the image cells (cells corresponding to `XX1`).
- State cell transition probabilities are updated:

  $$p^a_{s',s}\left(k+1\right)=\frac{np^a_{s',s}\left(k\right)+1}{n+1}$$

  where n is the number of previous evaluations of the state equation from cell s. Initially, $p\left(0\right)=0$. The n's are increased by one. Sink cell (if exists) is always the S'th cell, with $p^a_{S,S}=1$.
- Cell likelihoods are updated (the state measurement equation `funh` is evaluated for each state `XX`, the corresponding discrete measurement cells are located, and the transition probabilities are updated in a similar way as state cell transitions. Sink cell always maps to the M'th measurement cell, $l_{M,S}=1$.

Some information on model structure, execution phase, observed and estimated update times, and model stochasticity is displayed along the way.

Example 32 The model stored in vdvdata10 is based on ten evaluations at each cell. The model can be updated once using

```
Plant = vdv_plant_setup;
Plant = gcm_generate_model(Plant);
```

The CFMC transition probabilities are updated one-by-one for each control action ($a = 1, 2, ...$). During the evaluation for $a = 1$, statistics of elapsed times and estimates of remaining times are displayed:

```
GCM_GENERATE_MODEL: Initializing 60 sparse 34273x34273 matrices
with 205638 nonzeros...ok
GCM_GENERATE_MODEL: generating 1st CFMC model
17x14x6x6x4 grid structure assumed for Xref (S=34272).
Simulating plant model vdvf5, vdvh..
-Setting xfrom...( 0) ok.
-Evaluating xto...( 1:25) ok.
-Counting state transitions...( 0) ok.
-Updating state transition probabilities...( 6) ok.
-Updating measurement likelihoods...( 0) ok.
60 map updates ready in 1:30:28:  a=1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45
,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60
N = [1,1], off-diags 0.93 ( 1:27:04).  ok
```

The first two and the fourth row are displayed only at the first evaluation round, the evaluation round is indicated at the third row. The remaining info indicates that:
· DSS model M-files are `vdvf5` and `vdvh`
· generation of plant inputs (random points within cell hypercube) took 0 seconds
· evaluation of DSS equations (for the 34273 cells) took 1 min 25 seconds
· observing cell transitions took 0 seconds
· update of cell transition probabilities took 6 seconds, and
· updating measurement likelihoods to zero seconds.
· an estimate of the remaining time (for the 59 other actions) is then given (one and a half hours).

At the end of update, the following info is displayed
· the number of state evaluations is in the range [1,1],
· averaged over all P, 93% of mappings are off-diagonal (map to some other cell)

Uses: gcmdisp, gcm_check_for_grid, gcm_sampleu, gcm_sec2time, gcm_get_partitions, gcm_x2s

3.3.23 gcm_get_partitions

Find and sort all different elements in a vector

```
pvect = gcm_get_partitions(vect)
```

where `vect` is a vector with any elements, `pvect` is a list of different finite elements in vect ordered from smallest to largest.

This function is used extensively throughout the toolbox. Any changes to it should be done with care.

Example 33 `gcm_get_partitions([-1 -1 2.5 2.5 3])` returns `[-1 2.5 3]`. Also `gcm_get_partitions([2.5 -1 -1 2.5 3 Inf])` returns `[-1 2.5 3]`.

3.3.24 gcm_inigrid

Initialize a grid structure

```
Xref=gcm_inigrid(In1,In2,...,InX)
```

where `In1`, `In2`, ... are lists of elements in corresponding dimensions of the grid. `Xref` is a matrix containing all combinations of the elements (i.e., a full grid). This command is useful when initializing reference points.

An additional output argument will return a vector of input dimensions.

Example 34 The command `[Xref,ns] = gcm_inigrid([1 2],[3 3.5 4])` returns

$$\texttt{Xref} = \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 3 & 3.5 & 4 & 3 & 3.5 & 4 \end{bmatrix} \text{ and } \texttt{ns} = [2,3].$$

Uses: gcm_allpnts_int

3.3.25 gcmoptipol

Solve for the optimal control policy using value iteration:

```
Cost = gcmoptipol(Plant,Cost)
```

where the `Plant`-structure speficies the plant to be controlled, the `Cost`-structure contains plant controller specifications.

The following fields in `Cost(q)`'s are required (for each set point q):

- R, $S \times A$ matrix of immediate costs $r_{s,a}$
- gam, discount factor γ

The following fields the Cost(q) structures are updated:

- Jstar, optimal costs-to-go
- Pi, optimal policy π_s

See also gcmccp.

The function works as follows. For each set point at a time, a loop is constructed where the Q-factors for each state and control action are updated, the optimal policy cost-to-go and policy are obtained, and a termination criterion is cheked. If the termination criterion is not passed, a new iteration of the Q-factors is started.

Example 35 The command

```
load vdvdata10;
Cost = gcmoptipol(Plant,Cost,1)
```

solves the optimal policy for the first set point controller (stored in Cost(1)). The following information is displayed

```
GCMOPTIPOL: Solving a 34273x60 Bellman eq.  (q=1).
Value iteration:  until max | [J*(k)-J*(k-1)]/mean(J(k-1))] | < 1%:
0.958% ok.
```

This indicates that the criterion for exiting value iteration was 1% , and that the criterion at exit was 0.958%. Re-running the function, the value iteration is always performed at least once, so that repeating the command will give an exit condition of 0.931%.

Uses: gcm_automeq

3.3.26 gcm_pinf

Compute mapping to the stationary distribution

$$\texttt{P\_inf} = \texttt{gcm\_pinf}(\texttt{P})$$

- `P_inf` is a $S \times S$ mapping to a stationary distribution

The function computes the mapping $\mathbf{P}^{\infty}$ to a stationary distribution $\boldsymbol{\pi}^{\infty}$ given an initial state $\mathbf{p}_0$:

$$\mathbf{p}^{\infty} = \mathbf{P}^{\infty}\mathbf{p}_0.$$

$\mathbf{P}^{\infty}$ is a symmetric matrix with nonzeros only at rows/columns corresponding to absorbing/recurrent cells. It is computed by

$$\mathbf{P}^{\infty} = \frac{1}{n}\sum_{k=1000}^{1000+n} \mathbf{P}(k)$$

where n is chosen such that the largest absolute change in any element of $\mathbf{P}^{\infty}$ between updates $n-1$ and n will be less than 0.0001.

The above approach fails for large $\mathbf{P}$, due to memory problems. In such a case, a second strategy is automatically adopted. It consists of handling separately the mapping between recurrent cells and the mapping from transient to recurrent cells. Recurrent cells are handled by computing a time average far in the future (as above, $n = 100$). The mappings from transient to recurrent cells are included only if larger than a given tolerance. The second strategy uses `gcm_eiganalysis`. Results from a previous eigenanalysis can be included using additional input arguments `P_inf` = `gcm_pinf(P, EA, P_t2r)`.

Example 36 Consider the $\mathbf{P}$ matrix below (see example on `gcm_comm`). The command `P_inf = gcm_pinf(P)` will produce the mapping $\mathbf{P}^{\infty}$ to the stationary distribution:

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{P}^{\infty} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

For example, for an initial state cell distribution $\mathbf{e} = [0, 0, 1, 0, 0]^{\mathsf{T}}$,the stationary distribution is obtained from $\mathbf{P}^{\infty}\mathbf{e}$, and is $[0, 0, 0.5, 0.5, 0]^{\mathsf{T}}$.

Uses: gcm_eiganalysis, gcm_spsize, gcm_squeezep

3.3.27 gcm_plotx

Plot one, two or three vectors of sampled values showing discretization at y-axis:

```
gcm_plotx(Ts,Yref,Y)
gcm_plotx(Ts,Yref,Y,Y2)
gcm_plotx(Ts,Yref,Y,Y2,Y3)
```

where `Ts` is the sampling time, `Yref` is a row vector containing the discretization, and `Y` (`Y2`, `Y3`) is the data column vector.

Example 37 The following example plots the measured plant output and the set point trajectory. The discretization for plant output in the CFMC model is shown as y-axis ticks (emphasized by the grid-command):

```
load vdvdata10;
gcm_plotx(Plant.Ts,Plant.Yref,Sim.YY_m,Sim.WW);
set(gca,'ygrid','on'); xlabel('time'); legend({'w','y'});
```

The resulting plot is shown in Fig. 4.

Uses: gcm_get_partitions

3.3.28 gcm_pplot

Plot a trajectory of cell probabilities:

```
gcm_pplot(pp)
```

where `pp` is a $S \times N$ matrix of state cell probabilities (S is the number of cells, N is the number of steps in the trajectory). `gcm_pplot` produces an image where the probability of each cell at a given time step is illustrated by a colored rectilinear patch. The colorbar is shown in the image.

Example 38 The following command simulates the closed-loop cell trajectory from initial cell 1114 using the 3'rd action, projects the results to the first dimension, and plots the result:

```
load vdvdata10;
pp = gcm_step(gcmclosed(Plant,Cost),1114,3);
clf;
```

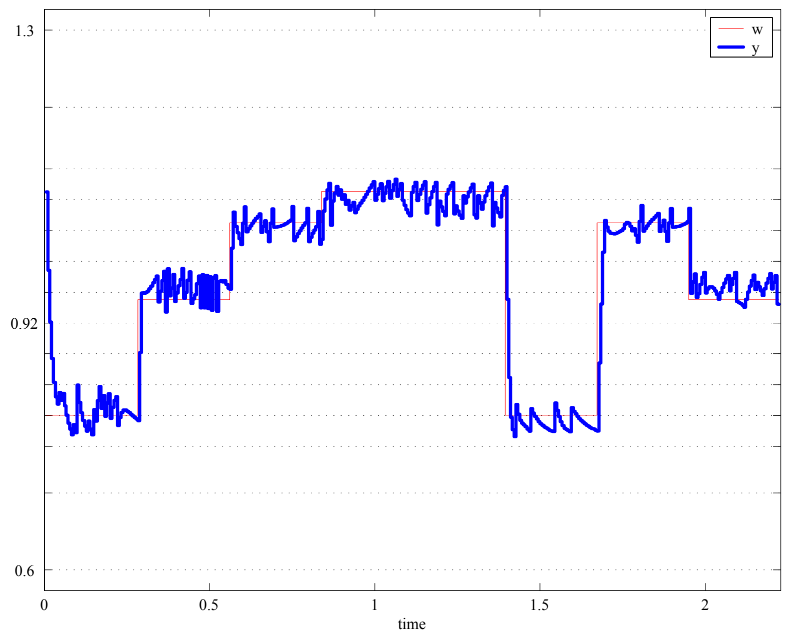


Figure 4: gcm_plotx

```
gcm_pplot(gcm_project(pp,Plant.Xref,1));
```

The resulting plot is shown in Figure 5.

3.3.29 gcmpred_online

Compute online the output of a predictive controller

```
a=gcmpred_online(p_x,Plant,Cost,q)
```

where the following information is required

- `p_x`, the (estimated) distribution of the state cell probabilities
- `Plant.f(a).P` for all `a`, the plant probability transition matrices for each control action
- `Cost(q)` with fields `H_p` and `R`, the controller control horizon and immediate costs

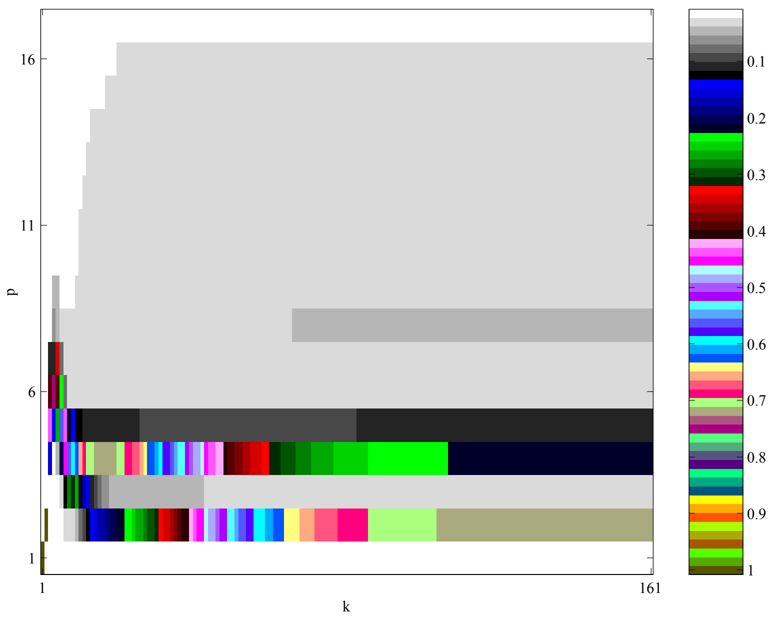


Figure 5: gcm_pplot

- `q`, the set point to be considered

The control action `a` is returned, i.e. the plant controller output should be set to `Plant.Uref(:,a)`.

The control horizon is assumed to be 1.

The function works as follows: The plant probability trajectories (in a prediction horizon from 0 to H_{p}) are solved for each control action a. The sums of immediate costs in the horizon are computed, weigted according to state cell probabilities. The action (a) resulting to a smallest cost is the controller output.

The code is adopted from `gcmpredpol`. See `gcmpredpol` for details.

Uses: gcn_squeezep, gcm_sec2time

3.3.30 gcmpredpol

Compute optimal policy using predictive control

```
Cost = gcmpredpol(Plant,Cost)
```

where the following information is required

- `Plant.f(a).P` for all `a`, the $S \times S$ plant probability transition matrices for each control action
- `Cost(q)`, for all `q` with fields `H_p` and `R`, the controller control horizon and $S \times A$ matrix of immediate costs

The following fields in `Cost(q)` vector structure are updated (for all set points `q`):

- `Pi`, the control policy, π_s^q
- `Jstar`, the optimal cost-to-go (in the prediction horizon)

If an additional input argument `q` is given, the only the q'th set point controller is computed.

The control horizon is assumed to be 1.

See also `gcmccp`.

The function is based on computing predictions for all initial states in one batch, i.e, predictions are computed for all possible 'certain' cells, but not for uncertain cells (for uncertainty in state cells, see `gcmpred_online`): For each control action $a = 1, 2, ..., A$, state probability matrix is initially set to unity matrix (certain states). For each step in the prediction horizon from zero to H_p, do the following: For each set point $q = 1, 2, ..., Q$, compute immediate costs weighted by the state cell probabilities, and sum it with costs from previous horizons. If size of the state probability prediction matrix is large, set small probabilities to zero. Using CFMC model, predict the state probability matrix for the next sampling instant. When the horizon is covered, take the next control action and repeat the process. Information on code execution progress is displayed along the way. Finally, optimal actions are found and fields `Pi` and `Jstar` are set.

Example 39 The commands

```
load vdvdata10;
Cost = gcmcpp(Cost,5);
Cost = gcmpredpol(Plant,Cost,1)
```

solve the predictive control policy for the first set point controller (stored in `Cost(1)`), using a prediction horizon 5. The following information is displayed:

```
GCM_PREDPOL: Evaluating set points q=1 for H_p = 5
xx2%xx3%xx5%xxxxxxxx12%xxxxxx17%xxxxxx
22%xxxxxx27%xxxxxx32%xxxxxx37%xxxxxx
42%xxxxxx47%xxxxxx52%xxxxxx57%xxxxxx62%xxxxxx
67%xxxxxx72%xxxxx77%xxxxxxxx83%xxxxx92%xxxxx
100% 100% ( 5:36) ok.
```

The percentages show the progress in computing step predictions for the A control actions. The 'x'-marks indicate that the size of the prediction matrix was decreased (using `gcm_squeezep`) when computing the predictions. In the example case it took 5 min 36 seconds to solve the predictive control policy.

Uses: gcm_squeezep, gcm_spsize, gcm_sec2time

3.3.31 gcmpredpoldp

Compute optimal policy using predictive control using dynamic programming (with control horizon as a tuning parameter):

```
Cost = gcmpredpoldp(Plant,Cost)
```

where the following information is required

- `Plant.f(a).P` for all `a`, the plant probability transition matrices for each control action
- `Cost(q)`, with the following fields for all `q`
  - `H_p`, the prediction horizon
  - `H_c`, the control horizon (by default `H_c` $= 1$)
  - `R`, the immediate costs

The following fields in `Cost(q)` vector structure are updated (for all set points `q`):

- `Pi`, the control policy, π_s^q
- `Jstar`, the optimal cost-to-go (in the prediction horizon)

If an additional input argument `q` is given, the only the q'the set point controller is computed.

The code uses dynamic programming, and solves the policy backwards (from last decisions to first decisions).

The code works as follows: Each set point `q` is considered separately. At stage $H_\mathrm{c} - 1$, predict future state probability trajectories for all S possible certain states, for all A constant control actions, and evaluate the costs by summing the immediate costs weighted by the state cell probabilities. For stages $H_\mathrm{c} - 2$ to 0, evaluate costs using action a (immediate costs plus the cost to go at the next stage, weighted by the transition probabilities), starting at stage $H_\mathrm{c} - 2$. The optimal action at a stage is the one that minimizes the costs. The optimal policy, returned at `Cost(q).Pi`, is the policy at stage zero.

3.3.32 gcm_project

Project simulated state cell trajectory towards a given dimension:

$$[\mathtt{ppp}, \mathtt{In}] = \mathtt{gcm\_project}(\mathtt{pp}, \mathtt{Xref}, \mathtt{ip})$$

where `pp` is a $S \times N$ matrix of state cell probabilities (S is the number of cells, N is the number of steps in the trajectory), `Xref` is a $n_\mathrm{X} \times S$ matrix of reference cells, and `ip` is the projection direction (`ip`$\in \{1, 2, ..., n_\mathrm{X}\}$. `ppp` will have as N columns and as many rows as there are distinct values in the `ip`'th row of `Xref`.

Example 40 Let pp and `Xref` be 5×3 and 2×5 matrices ($S = 5$, $N = 3$) given by

$$\mathtt{pp} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0 \\ 0 & 0.5 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \mathtt{Xref} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 6 & 6 & 7 & 7 \end{bmatrix}$$

Projection to 1'st dimension of `Xref`:
`[ppp1,In1]=gcm_project(pp,Xref,1)`
results in `ppp1` equal to `pp` (since all elements in the first row of `Xref`, corresponding to values in `In1` $= [1, 2, 3, 4, 5]$, are distinct).

Projection towards the 2'nd dimension of `Xref`
`[ppp2,In2]=gcm_project(pp,Xref,2)`
gives a $2 \times N$ vector:

$$\texttt{ppp2} = \begin{bmatrix} 1 & 0.5 & 0 \\ 0 & 0.5 & 1 \end{bmatrix}$$

corresponding to values in `In2` $= [6, 7]$.

Uses: gcm_get_partitions

3.3.33 gcm_rmse

Compute root-mean-squared-error and mean error between two vectors

```
[r,m] = gcm_rmse(Yhat,Y)
```

where `Yhat` and `Y` are column vectors of same size. The first output argument is the rmse, the second is the mean error:

$$r = \sqrt{\frac{\sum_{k=1}^{K} \left[\widehat{y}(k) - y(k)\right]^2}{K}}, m = \frac{\sum_{k=1}^{K} \left[\widehat{y}(k) - y(k)\right]}{K}$$

- If `Yhat` is a matrix and `Y` is a vector, each column is compared with the vector.
- If `Yhat` and `Y` are matrices of same size, vectors at the same columns are compared with each other.

Example 41 Define matrices **U** and **V** as follows (compare with `gcm_eucl_norm`):

$$\mathbf{U} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{V} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

The following commands compute RMSE between two vectors, a vector and a three column vectors, and between respective columns of the three column vectors

```
gcm_eucl_norm(V(:,2),U(:,1))
```
returns $R = 1.29$.

```
E=gcm_eucl_norm(V,U(:,1))
```
returns a vector:

$$\mathbf{R} = [\ 0 \quad 1.29 \quad 1.83\ ]$$

```
E=gcm_eucl_norm(V,U)
```
returns a vector:

$$\mathbf{R} = [\ 0 \quad 0.58 \quad 1.15\ ]$$

3.3.34 gcmrun

Run successive model updates with occassional controller updates

```
gcmrun
```

Assumes that a properly initialized `Plant` structure (named as `Plant`) exists in the MATLAB workspace. For required info, see `gcm_generate_model`. In subsequent updates, the `Plant` structure (named as `Plant`) is updated to the workspace. The maximum number of plant evaluations is determined from variable `EVA` in workspace (by default `EVA` is set to 500). After each update, the results are stored in a file named as `gcmrun.mat`.

If a `Cost` structure (named as `Cost`) exists in the workspace, an optimal or predictive controller for the plant is designed at evaluations 1,2,3,10,50 and after every 100'th evaluation, and the `Cost`-structure is updated. Similarly, if a `Sim`-structure exists (named as `Sim`), a simulation of the closed loop system is performed and the `Sim` structure is updated.

Example 42 The following command continues the updating of the existing CFMC model:

```
load vdvdata10
gcmrun
```

Note that from `vdvdata10`, the CFMC model exist in the MATLAB workspace in a structure named as `Plant`. Since structures `Cost` and `Sim` also exist in `vdvdata10`, the plant controller is recomputed and the closed-loop system is simulations are performed every now and then. Between successive model updates, the following info is displayed:

```
=== 28-Nov-2007 12:16:03 === EVA = 11 of 500:  ==============
```

showing the date, clock and the values of current and maximum number of plant evaluations (here: `11 of 500`).

Uses: gcm_generate_model, gcmoptipol, gcmpredpol, gcmsim

3.3.35 gcm_sampleu

Generate random samples from a uniform distribution within a supporting hypercube:

```
Xrnd = gcm_sampleu(Xref,In)
```

`Xref` is a matrix of S column vectors. An optional input argument `In` can be given, which is a cell array whos elements are row vectors describing the partitioning for the respective dimension. This presupposes that `Xref` is obtained from a grid. If `In` is not given, it is obtained using `gcm_get_partitions`.

- `Xrnd` returns one random sample from the support of each vector in `Xref`. The supports of the elements at the edges are assumed to be symmetric.

Example 43 Let us define a partitioning of using four reference points: `In{1}=[1,2,3,4]`. We can now generate random samples for 1'st and 3'rd elements as follows

```
gcm_sampleu([1,3],In)
```

which results in a vector `[0.6782 3.0894]` (1'st $\in [0.5, 1.5]$, 3'rd$\in [2.5, 3.5]$).

Example 44 Let us generate random samples from the hypercube associated with cell 1114:

```
load vdvdata10;
[YN,In]=gcm_check_for_grid(Plant.Xref);
Xrnd = gcm_sampleu(Plant.Xref(:,1114),In)
```

Uses: gcm_get_partitions

3.3.36 gcm_sec2time

Convert seconds to a time notation string. The commands

```
str  =  sec2time(sec)
str  =  sec2time(sec,t0)
```

produce a string `str` in the format hh:mm:ss dd/mm/yyyy. A second input argument is a date vector to which `sec` seconds is added. An additional output argument returns the date vector corresponding to the string `str`.

The function does not work over month boundaries.

Example 45 The following table illustrates the command

command	result
`gcm_sec2time(70)`	`1:10`
`gcm_sec2time(3600)`	`1:00:00`
`gcm_sec2time(0,clock)`	`12:19:34 20/11/2007`
`gcm_sec2time(70,clock)`	`12:20:44 20/11/2007`

3.3.37 gcmsim

Run a simulation:

```
Sim = gcmsim(Plant,Cost,Sim)
```

where the `Plant`-structure speficies the plant to be controlled, the `Cost`-structure specifies the plant controller, and the `Sim`-structure specifies the simulation details. Simulation results are stored in the `Sim`-structure output argument.

The Sim-structure contains information for making a sample simulation, and results of a simulation. In order to start a simulation (using `gcmsim`) additional information on plant initial state and state estimator are needed, as well as a desired target trajectory. As results of the simulation, the plant input – output trajectories are stored, various state trajectories (real, measured, discretized), as well as some statistical information (rmse on output, etc.).

The following fields are required in `Sim`:

- `WW` is a K x n_{y} matrix of the target set points, each set point as a row vector

- `x0` is a column vector of plant's initial state (simulated plant)
- `y0` is a column vector of plant's initial output (simulated plant)
- `p_x0` is the initial state distribution (plant model), $S \times 1$

The following fields are created/updated:

- `UU` is the K x n_{u} matrix containing the control sequence, where each row speficies the control action applied at each sampling instant.
- `XX_s`, `XX_m`, `XX_d` are state sequences for simulated (noiseless), measured and discrete (estimated) state sequences.
- `YY_s`, `YY_m` and `YY_d` are the corresponding measurement sequences.
- `PDFX` is a matrix of state distributions, $K \times S$
- `QMSA` is a matrix with the following rows [q,m,s,a] corresponding to set point, measurement, state and control action indexes at each sampling instant.
- `rmse_rd` is a $K \times 2$ matrix with RMSE on measured (first column) and RMSE on discrete output (vs. target set points).
- `mctg` is the mean cost to go

The `gcmsim` function works as follows: First, the state and measurement of the simulated plant are set according to `Sim.x0` and `Sim.y0`. The system state and measurement are estimated using `gcm_estimate_x` and `gcm_estimate_y` (assumed to be measurable, see code for alternatives). The next set point is obtained from `Sim.WW`, and the corresponding controller is chosen. Based on a maximum likelihood state estimate, a control action is chosen from the control policy in `Cost.Pi` (see code of `gcmsim` for an on-line predictive controller based on a state distribution estimate). The action is applied to the simulated plant, and the plant is simulated one step ahead. The `Sim`-fields are updated and the simulated trajectories are plotted. The program flow then retunrs to state estimation, until the end of the setpoint trajectory is reached.

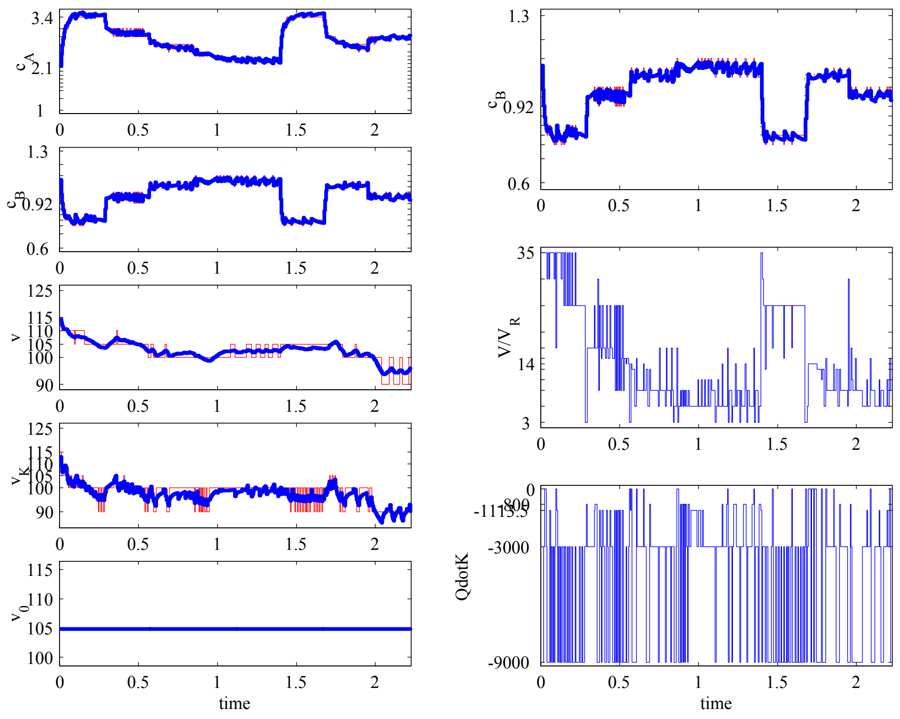


Figure 6: gcmsim (gcm_sim_plot)

Example 46 The following command updates the Sim-structure with simulation information:

```
load vdvdata10;
Sim = gcmsim(Plant,Cost,Sim)
```

A plot of simulation results is also produced, see Fig. 6 (see also gcm_sim_plot).

Uses: gcm_sim_plot, gcm_estimate_x, gcm_estimate_y, gcm_x2s, gcm-pred_online

3.3.38 gcm_simp

Simulate a probability transition model

```
pp = gcm_simp(P,p0)
```

where `P` is the $S \times S$ probability transition matrix and `p0` is the initial state distribution, $S \times 1$, or a scalar index s to a sure state. `pp` is a $S \times (k_\mathrm{f} + 1)$ matrix containing the state cell probabilities in each column, where k_f is the simulation length. k_f is determined automatically, or can be given as an additional input argument:

```
pp = gcm_simp(P,p0,kf)
```

If the output argument is not given, a plot of the state probabilities is constructed.

Example 47 Consider the $\mathbf{P}$ matrix and initial state $\mathbf{p}_0$ below

$$\mathbf{P} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{p}_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

The simulation of the system using

```
pp = gcm_simp(P,p0);
```

results in a sparse matrix where the k'th column indicates the cell probabilities at the k'th time step (initial state is $k = 1$):

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Uses: gcm_ff.

3.3.39 gcm_sim_plot

Plot results of a simulation (x,u,y):

```
gcm_sim_plot(Plant,Cost,Sim)
```

This command produces a plot of the simulation, as stored in the Sim-structure. The state trajectories are plotted to the left of the plot, the outputs to the top right corner and the control inputs at the bottom right corner. The trajectory of each variable is shown in its own plot.

The plots generated by the commands

```
load vdvdata10;
gcm_sim_plot(Plant,Cost,Sim);
```

is illustrated in Figure 6.

Uses: gcm_plotx, gcm_ff

3.3.40 gcm_sim_plot_io

Plot results of a simulation (u,y):

```
gcm_sim_plot(Plant,Cost,Sim)
```

This command produces a plot of the simulation, as stored in the Sim-structure. The plant outputs are shown in the top and the control inputs at the bottom. The trajectory of each variable is shown in its own plot.

Example 48 The plots generated by the commands

```
load vdvdata10;
gcm_sim_plot_io(Plant,Cost,Sim);
```

is illustrated in Figure 7

Uses: gcm_plotx, gcm_ff

3.3.41 gcm_spsize

Get information of the size of a sparse variable:

```
[nz,n,m]=spsize(M)
```

where `M` is a $n \times m$ sparse matrix of size, `nz` is an approximate estimate of the number of nonzeros.

The estimate is based on examation of the bytes-information from the command `whos`:$nz = \max\left(\frac{\texttt{bytes}-64}{12}, 20\right)$. It is useful in cases when a direct

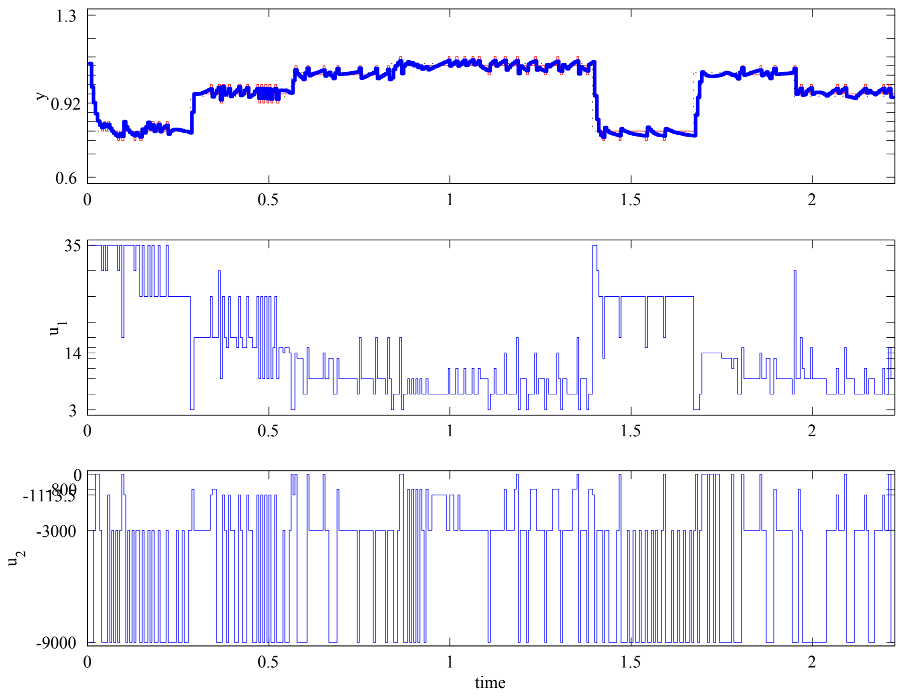


Figure 7: gcm_sim_plot_io

evaluation (e.g., counting nonzero elements) produces an out-of-memory error.

Example 49 The estimate is a rough upper bound estimate, as shown by the following test-script:

```
load vdvdata10;
for i=1:100,
S=342*i
[spsize(Plant.f(1).P(1:S,1:S)), sum(sum(P(1:S,1:S)~=0))]
end
```

which produces the results shown in the table below

S	true nz	estimated nz
324	946	1280
3420	13073	14829
34200	157652	168958

3.3.42 gcm_squeezep

Set small elements in a sparse matrix to zero, ensuring preservation of probability measure

```
pp_out   =   gcm_squeezep(pp,sqlimit)
pp_out   =   gcm_squeezep(pp,LIM)
```

where `pp` is a $S \times N$ column matrix of N probability distributions. If the second input argument (`sqlimit`) is less than 1, all probabilities less than `sqlimit` are set to zero. The remaining probabilities are normalized so that the probability measure is preserved. A maximum value for `sqlimit` is 0.1.

If the second input argument is greater than 1 (`LIM`), an sqlimit is solved such that the `pp_out` will contain no more than `LIM` nonzeros. If only one input argument is given, `sqlimit` is set to $\frac{1}{S}$.

This code is needed for reducing the size of large sparse matrices (to avoid out-of-memory problems). For the same reason, the code is written in batches. The use of `LIM` is very approximative..

Example 50 Let a matrix $\mathbf{P}$ be given as follows:

$$\mathbf{P} = \begin{bmatrix} 0.6273 & 0.4952 & 0.3353 \\ 0.0539 & 0.1741 & 0.1496 \\ 0.2020 & 0.2124 & 0.2378 \\ 0.1168 & 0.1183 & 0.2773 \end{bmatrix}, \mathbf{Q} = \begin{bmatrix} 0.6630 & 0.4952 & 0.3353 \\ 0 & 0.1741 & 0.1496 \\ 0.2135 & 0.2124 & 0.2378 \\ 0.1235 & 0.1183 & 0.2773 \end{bmatrix}$$

where $\mathbf{Q}$ is obtained from applying the command `Q=squeezep(P,0.07)`.

3.3.43 gcmstability

Analyse stability of a (closed-loop) system

```
[Sink,Stat]  =  gcm_stability(Plant_cl)
[Sink,Stat]  =  gcm_stability(Plant,Cost)
```

where `Plant_cl` is the closed loop `Plant`-structure, containing the closed loop CFMC model. A cell c is considered to be stable if the static probability of entering the sink cell, given an initial cell c, is zero.

With two input arguments, a closed-loop map is computed from plant and controller descriptions. The function returns two output arguments:

- `Sink`, is a $Q \times S$ matrix giving the stationary probability of entering the sink cell, when initially in cell s and when under set point control q.
- `Stat` is a vector structure, where `Stat(q)` has two fields: `P_inf`, the stationary closed loop map, and `EA`, a structure whos fields contain indexes to recurrent and transient cells (see `gcm_eiganalysis`).

If no output arguments are given, a bar plot of the state distributions is produced, one plot for each set point q. Each bar shows the percentage of stable initial cells (among all state cells). This plot can also be evoked by using the `Sink` matrix as an input argument:

`gcm_stability(Sink)`

With additional input arguments, `gcm_stability(Sink, [], ss)`, the plot restricts to only cells in the list `ss`.

Example 51 The stability of the closed-loop system can be examined by first computing the `Sink`-matrix, and then plotting the results:

```
load vdvdata10;
[Sink,Stat] = gcmstability(Plant,Cost);
gcmstability(Sink);
```

The generated plot is shown in Figure 8. It can be observed that the system is stable in almost initial cells. There are 1+24 cells from which the probability of entering is nonzero (probability between 0 and 0.5). The one unstable state (with $p = 1$) is the sink-cell itself.

Further examination of the `Sink` matrix reveals that with all setpoints, the possibly unstable initial states are the same (`ius = find(Sink(1,:)>0)`). Using `gcm_dispref(ius,Plant.Xref)`, gives

```
A set of 26 states:
x(1) = { 3.1 , 3.4*, Inf }
x(2) = { 1.2 , 1.3*, Inf }
x(3) = { 125*, Inf }
x(4) = { 90 , 100 , 105 , 110 , 115 , 125*, Inf }
x(5) = { 100 , 105 , 110 , 115*, Inf }
```

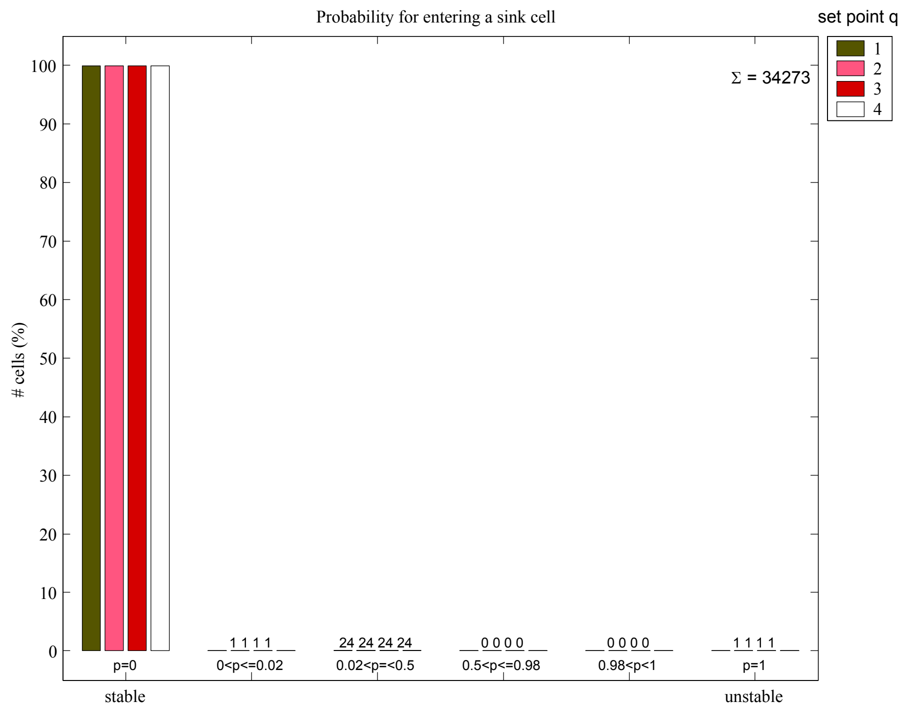


Figure 8: gcmstability

and we conclude that the unstabilities are associated with high values for states $x_1 - x_3$ (high concentrations and temperature of the vdv process).

Uses: gcmclosed, gcm_pinf, gcm_ff

3.3.44 gcm_step

Simulate and plot a discete state trajectory from a given initial state

```
gcm_step(Plant,s,a)
gcm_step(Plant,p0,a)
```

where `Plant` is a sturcture containing the system CFMC model, `s` (`p0`) is the initial state cell (distribution), and `a` is the constant control action applied to the plant.

An additional argument `T` can be given to control the time scale of the simulation: `gcm_step(Plant,s,a,T)`. If `T` is a scalar, it is the end-time of the simulation (in real time, vs. sampling time given in the `Plant_cl` or `Plant`-structure). If `T` contains two elements, the first is the start time, the second is the end-time of simulation. If `T` is a vector, the corresponding points in time are returned (sampling time should be the same as in the model). By default, simulation is continued until a steady state is observed (or 1000 steps are exeeded). The generated plot is a function of samples (not real time).

A further additional argument `ilist` can be given to specify the states for which the trajectory is to be plotted: `gcm_step(Plant,s,a,T,ilist)`. Default is all states (`ilist = [1:size(Plant.Xref,1)]`).

If an output argument is given, a matrix of state probability transitions for all states is returned.

`gcm_step` uses `gcm_simp` to simulate transitions of probability distributions.

Example 52 In the following example, `gcm_step` is used to generate a plot of the propagation of the state cell probabilities, projected towards the second state:

```
load vdvdata10
Plant_cl = gcmclosed(Plant,Cost);
gcm_step(Plant_cl,1114,3,[],2);
```

Resulting plot is shown in Figure 9.

Uses: gcm_project, gcm_pplot, gcm_get_partitions, gcm_ff.

3.3.45 gcm_v2v

Convert from vector to matrix and vice versa:

$$\begin{aligned} \texttt{V} &= \texttt{gcm\_v2v(M)} \\ \texttt{M} &= \texttt{gcm\_v2v(V,r)} \end{aligned}$$

where `M` is a $r \times c$ matrix and `V` is a $rc \times 1$ vector.

Example 53 Define $\mathbf{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. Then `V=gcm_v2v(M)` and `N=gcm_v2v(V,2)` result in $\mathbf{V} = [\ 1 \quad 2 \quad 3 \quad 4\ ]^{\mathsf{T}}$ and $\mathbf{N} = \mathbf{M}$.

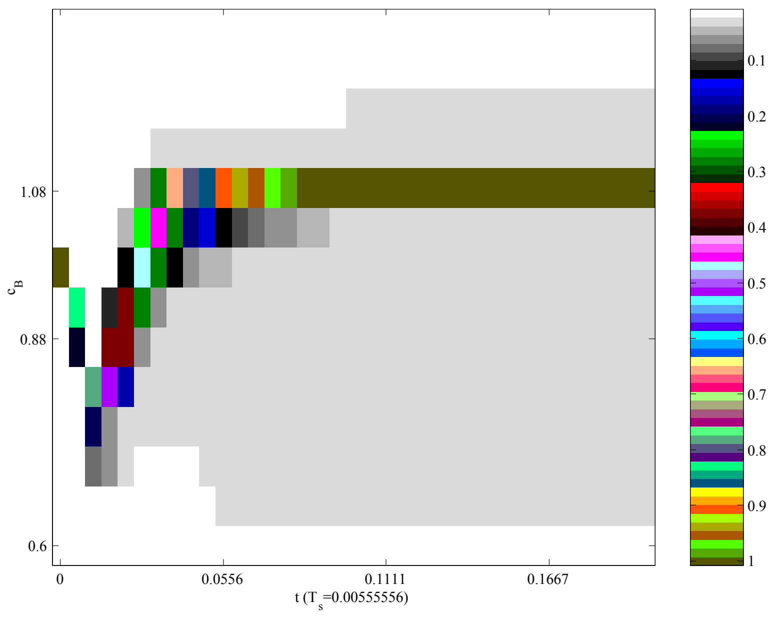


Figure 9: gcm_step

3.3.46 gcm_vodesolver

One step ahead solver for ode in a vectorized form. The syntax of the command is

```
Xk1B = gcm_vodesolver(function_handle,XkB,Uk,Ts)
```

where `function_handle` is the handle to the ode M-file, `XkB` is a column matrix of input state vectors (at time t), `Uk` is a control vector (constant between t and t+`Ts`), `Ts` is the sampling time. `Xk1B` is matrix of states at the next time instant (t+`Ts`). The ode are solved using `ode23t`.

The point of the function is in that it solves the equations in small batches, as it seems to be the most efficient way.

Example 54 The following command solves the vdv-ode-equations one step ahead starting from state cell 1114's reference point:

```
load vdvdata10;
s = 1114; q = 3;
a = Cost(q).Pi(s)
X_next = gcm_vodesolver(@vdvdv5,Plant.Xref(:,s),Plant.Uref(:,a),Plant.Ts);
```

Note, that the `@vdvdv5` provides a handle to the ode-model (not the DSS model).

Uses: gcm_v2v

3.3.47 gcm_x2s

Find the closest cell corresponding to a state:

`ss = gcm_x2s(XX,Xref)`

where `XX` is a $n_x \times N$ column matrix of state vectors and `Xref` is a $n_x \times S$ column matrix of reference vectors. For each column vector in `XX`, the function finds the index to a reference vector which is closest to the vector. The indexes are returned in the S-row vector `ss`.

If an additional input argument is given:

`ss = gcm_x2s(XX,Xref,In)`

a grid structure is assumed. This results in much faster computations. `In` is an array structure, where each element `In{i}` specifies a vector of discretized values for the i'th dimension (see `gcm_check_for_grid`)

With two additional output arguments (`lb` and `ub`) the lower and upper bounds for each dimension are returned (useful if `Xref` contains a sink cell).

If any of the elements in `Xref` is not finite, it is assumed that the last cell is a sink cell.

This function is used throughout the toolbox files, so all modifications to it should be done with care.

Example 55 Define the reference set by

$$\texttt{Xref} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 6 & 6 & 7 & 7 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} 2.3 \\ 6.1 \end{bmatrix}$$

Index to the closest matching vector in the reference set is obtained with the command

```
s=gcm_x2s(x,Xref);
```

which gives $s = 2$.

Uses: gcm_chbase

3.3.48 unrav*

Unravelling algorithm, adopted from the book by C.S.Hsu (1987) "Cell-to-cell mapping" pp. 148-151.

- `unravel` % unravelling algorithm: find group, pediodicty and step numbers, etc., plot results
  - `unrav_ng` handle periodic motion
  - `unrav_og` handle previously known cases
  - `unravel_disp` plot results of the unravelling algorithm

3.3.49 vdv*

Sample files for the vdv5-example, see Section 3.1.

4 Future to-do items

The purpose of this report was to write out the current state of the software. A number of future enhancements can be considered. From coding point of view, a grafical user interface (GUI) would be very useful. A convenient way to deal with delays could also be sorted out (now all delays need to be included as additional states). Of course, interpretation of model and closed-loop analysis could be improved, in terms of powerfulness and ease-of-use. Items such as required model size (discretization) vs. computation time / memory requirements; required accuracy of open-loop map / # iterations; effect of constraints on reaching the optimal set point, etc. should be coded into the software. From research point of view, a number of interesting directions exist, including: handling of model uncertainty and accuracy; handling of noise/unmodelled dynamics (output error dynamics, integral errors (random walk), load disturbances); and learning and adaptation. Just to mention a few...