CCBM – A tool for activity recognition using Computational Causal Behavior Models

Thomas Kirste and Frank Krüger

May 24, 2012

Contents

1 Introduction **1**

2 A Toy Example **1**

3 CCBM Design Goals **3**

4 The CCBM System **4**
4.1 Domain Definitions 5
4.1.1 Domain File 5
4.1.2 Action Definitions 5
4.1.3 Formulae 6
4.1.4 Interfacing Observations to `C/C++` 7
4.1.5 Problem File 8
4.2 Tools 8
4.2.1 Compiler Script 8
4.2.2 Analyzer 8
4.2.3 Filter 9
4.2.4 Validator 10

5 Design Considerations **11**
5.1 Translation Strategy 11
5.2 Action Selection 13
5.3 Multi-Agent Sampling: the `:agent` slot 14
5.4 Simple Durative Actions: the `:duration` slot 16

6 Heavy Tails **18**
6.1 Compressing Probabilities 24
6.2 Resampling Using a Heavy-Tailed Density 25
6.3 Effect of Heavy-Tailed Resampling 30

1 Introduction

CCBM allows to describe the behavior of dynamic systems using symbolic precondition-effect rules. Based on such a *causal model* of the system dynamics, CCBM estimates the possible course of actions from a sequence of observation using Bayesian filtering. CCBMthus provides a specific *combination* of symbolic and probabilistic inference for the purpose of estimating the trajectories of dynamic systems from observation data.

The objective of CCBM is to enable the construction of probabilistic inference systems for those domains, where *prior knowledge* on the causality of system behavior is available. CCBM synthesizes probabilistic models from this prior knowledge, with the aim of reducing or ideally superseding the need for training data.

2 A Toy Example

As a toy example (see `Examples/XYRouter`), consider the following domain: A room of size 7m $\times$ 7m, a door at location $(2.1, 0)$ and a Ubisense position sensing tag at the wall, at position $(2.8, 0.7)$. A person enters, picks up the tag, and goes to the corner at position $(6.3, 6.3)$. From the location data provided by Ubisense, we want to estimate the route.

Clearly, the tool of choice in this simple case would be to use a Kalman-Filter. However, this setting is also a very easy situation for a causal model: We cover the room with a virtual grid of 5×5 cells, each cell being 1.4×1.4 meters in size. The causal model provides one action: moving from one cell to another cell. If the user is in a cell, we expect to observe him at the coordinates of the cell center, with a normally distributed error around this location. The mean for the expected location reading given that the user is at a specific cell is defined by the `:observation` clause.

```
(:action go (?from ?to - cell)
    :duration (exponential 0.1)
    :precondition (and (at ?from))
                       (connected ?from ?to)
    :effect (and (not (at ?from))
                 (at ?to)))
(:observations
    (forall (?c - cell)
            (when (at ?c)
                  (expectLocationReadingAt (x-pos ?c)
                                           (y-pos ?c))))))
```

Where the specific situation (cells, cell coordinates, cell connectivity, initial and final location), is given by:

```
(:constants 11 12 13 14 15
            21 22 23 .. 55 - cell)
(inits (= (x-pos 11 0.7))
       (= (y-pos 11 0.7))
       ...
       (= (y-pos 55) 6.3)
       (connected 11 12)
       (connected 11 21)
       ...
       (connected 45 55)
       (at 21))
(goal (at 55))
```

Observations for this model are read from a file containing one X/Y-coordinate pair (see `Examples/XYRouter/Data/r218-ur-xy.dat`), such as:

```
2.7974312 0.61454225
2.6715872 0.43351072
2.6644163 0.44700468
2.171642 0.5551207
…
```

Running the system – for instance by using the script `Examples/XYRouter/xyrouter1.sh` – provides as output:

```
…
Particle filter for domain: "xyrouter", problem: "xyrouter", …
…
*** Final log-likelihood: -159.809
    Finishing time:       75
…
*** Best successful particle: 10, weight: 0.000417602
Particle [19] history:
0: * (INITIALIZE)
10: * (step 21 22)
16: * (step 22 32)
22: * (step 32 42)
24: * (step 42 52)
27: * (step 52 53)
36: * (step 53 54)
46: * (step 54 55)
59: * (FINISHED)
75: (SUCCESS)
```

Which gives the the trajectory of the particle having the highest probability. In addition, the filter produces the produces the values $p(x_t \mid y_{1:t})$, giving for each time step the probability of each state. These probabilities have been used to compute the estimates (= mean positions) in the upper left plot of Fig. 1. As the

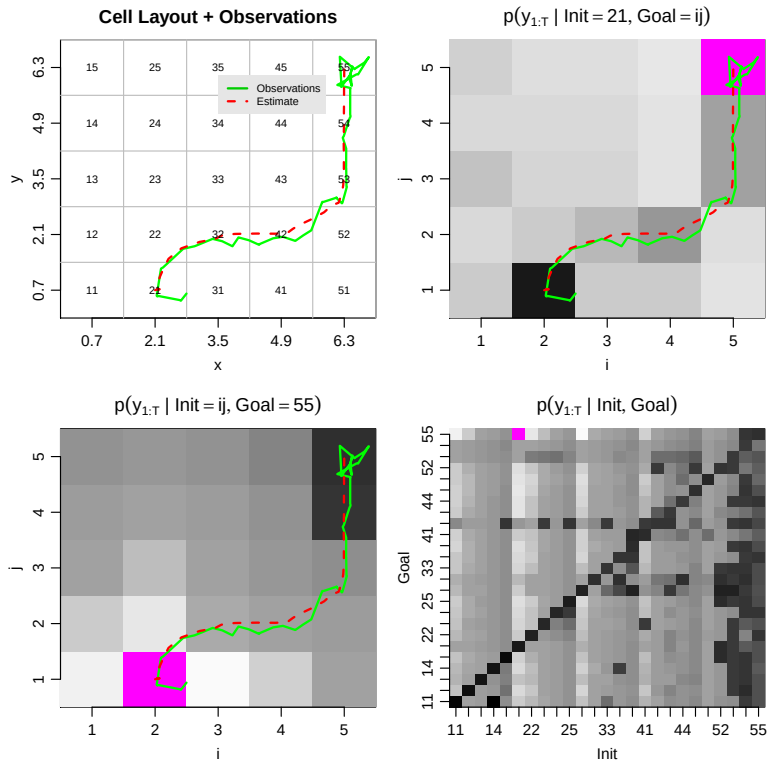

Figure 1: XYRouter domain, example plots

system also gives the final likelihood $p(y_{1:T})$, as well as the current likelihood values $p(y_{1:t})$, it allows to compare different models (*e.g.*, different domain definitions, different problem files), For instance, the likelihoods of different goals, different initial conditions, or different combinations of goal and initial condition have been compared by his method for the XYRouter domain be compared (see other plots in Fig. 1; brightness represents probability, magenta indicates state of higher probability).

3 CCBM Design Goals

Functionality and feature set of CCBM has been designed with the following goals in mind:

- Allow to infer goals (as labels *as well as* as desired states of the environment enabling the deliberate selection of intervention strategies)
- Allow to infer plans (as rational strategies – sequences of actions) that establish specific goals

- Allow to recognize individual actions (as "atomic" changes to the environment)
- Recognize environment states
- Predict actions (and, via model comparison, predict goals & plans)
- Enable decision-theoretic reasoning about system activities based on a quantification of the reliability of the estimates
- Allow for unreliable and noisy sensor data (probabilistic sensor models)
- Reduce requirements for training data (via a-priori models)
- Enable the use of symbolic causal modeling paradigms for building a-priori models
- Allow the development of reusable inference models
- Allow inference in (very) large state spaces
- Support inference in domains with multiple autonomous actors (multi-person scenarios)
- Allow flexible probabilistic duration densities
- Allow real-time inference
- Employ fully probabilistic inference (Bayesian filtering, smoothing, prediction, parameter estimation) (required for decision theoretic action selection)
- Support different / combined heuristics for action selection in order to adequately represent human problem solving strategies.

4 The CCBM System

The basic design concept of CCBM is to allow the definition of a symbolic causal domain model using a PDDL dialect, which is then translated into a probabilistic inference system using Monte-Carlo techniques for coping with virtually unlimited state spaces.

The translation process, invoked by the `rc` command of the CCBM system, translates the domain definition into a C-code module which is compiled and linked against the modules containing the filter routines. In addition, a user provided

C-code module containing the routines for computing the observation probabilities from observation data is compiled and linked into the final executable.

For building a filter, the following files have to be provided:

- Domain definition (types, constants, actions) [PDDL]
- Problem definition (problem constants, initial state, goal definition) [PDDL]
- Observation function [C]

4.1 Domain Definitions

4.1.1 Domain File

domain = `(define (domain` *name*`)`
`{` *domain-entry* `})`

domain-entry = `(:types {` *name* `{` *name* `} - ` *name* `} {` *name* `})`
| `(:predicates {(` *name* `[` *declarations* `]) })`
| `(:constants {` *constant* `{` *constant* `} - ` *type-name* `} {` *constant* `})`
| `(:observation {` *observation-formula* `})`
| *action*

type-name = *name*

declarations = `{{` *var* `} - ` *type-name* `} {` *var* `}`

4.1.2 Action Definitions

action = `(:action` *name*
`[:parameters (`*declarations*`) ]`
`[:saliency` *expression*`]`
`[:agent` *atomic-expression*`]`
`[:duration` *atomic-formula*`]`
`[:precondition` *formula*`]`
`[:effect` *effect-formula*`]`
`[:callbacks` *callback-list*`] )`

The `:callbacks` clause allows to tie callbacks to actions (just as the clause for states). At each time step, the callbacks of the actions for each agent will be called.

4.1.3 Formulae

Precondition formulae

formula = *atomic-formula*
| `(and` { *formula* } `)`
| `(or` { *formula* } `)`
| `(not` *formula* `)`
| `(imply` *formula* *formula* `)`
| `(iff` *formula* *formula* `)`
| `(forall` `(`*declarations*`)` *formula* `)`
| `(exists` `(`*declarations*`)` *formula* `)`

atomic-formula = *name*
| `(`*name* { *expression* } `)`
| `(=` { *expression* } `)`

Expressions

expression = *atomic-expression* | *fun-call*
fun-call = `(`*name* { *atomic-expression* } `)`
atomic-expression = *const* | *var*
constant = *name*
| *intConst*
| *floatConst*
var = `?`*name*

Effect formulae

effect-formula = *cond-effect*
| `(and` { *cond-effect* } `)`
cond-effect = *effect*
| `(forall` `(`*declarations*`)` *effect-formula* `)`
| `(when` *formula* `(`*effect* | `(and` { *effect* } `)` `)` `)`
effect = *atomic-formula* | `(not` *atomic-formula* `)`

Observation formulae

observation-formula = `(forall` `(`*declarations*`)` *observation-formula* `)`
| `(forall` `(`*declarations*`)` `(and` { *observation-formula* } `)` `)`
| `(when` *formula* `(`*observation* | `(and` { *observation* } `)` `)` `)`
observation = *atomic-formula*
callback = *atomic-formula* | *observation-formula*
callback-list = *callback* | `(and` { *callback* } `)`

4.1.4 Interfacing Observations to C/C++

Code for handling observations has to be provided in the observation header file submitted to the `rc` compiler script. (See the `-o` option of `rc`)

This header file should provide the following functions

- `void *fetchObservation(void);`

  This function is called once for every time step; it should read an observation y_t from standard input, store it wherever convenient, and return `NULL` in case of end-of-file.

- `double observe(StatePtr x);`

  This function should return the probability of seeing the last observation read by `fetchObservation` given the state `x`, that is: the value of $p(Y_t = y_t \mid X_t = \mathtt{x})$.

  `x` is a pointer to a bit vector representing the current state. As the mapping of propositions to bits is usually opaque to the implementer, the `:observation` declaration (see below) can be used to call specific C-functions in case of specific state configurations.

  The execution of these helper functions is initiated by calling `stateObservation(x)` at the beginning of `observe`.

In an observation such as

```
(:observation (forall (?p - person ?l - location)
              (when (at ?p ?l)
                    (setLocationSensor ?p ?l))))
```

the generated code will assume that there is a function

```
void setLocationSensor(Person p, Location l);
```

with suitably defined types `Person` and `Location`. The generator will insert the respective constant names in the call. So if there are constants `Max - person` and `Stage - location`, one of the generated calls will be

```
setLocationSensor(Max,Stage);
```

meaning that suitable `C/C++` values for `Max` and `Stage` have to be defined. For *intConst* and *floatConst* values, the `C/C++` types `long int` and `double` can directly be used.

4.1.5 Problem File

problem = (define (problem *name*)
{ *problem-entry* })

problem-entry = (:domain *name*)
| (:objects { *constant* { *constant* } - *type-name* } { *constant* })
| (:init [:duration *atomic-formula*] { *init-elem* })
| (:goal *formula*)
init-elem = *cond-effect*
| (= *fun-call constant*)

4.2 Tools

Core of the CCBM toolset consists of the `rc` compiler script that will, for a given domain definition, create a set of specialized executables for solving inference problems in this domain.

All tools provide brief usage information in reply to the `-?` option.

4.2.1 Compiler Script

Synopsis:

`rc` [*options*] `-d` *domain-file* `-p` *problem-file*

Generates

- State space analyzer, *problem*`-analyzer`
- Particle filter, *problem*`-filter`
- Validator, *problem*`-validater`

Note that the generator requires a suitable file *domain*`-observations.h` that defines the observation and estimation functionality. See `Examples/XYRouter/Models` for an example.

4.2.2 Analyzer

Synopsis:

problem`-analyzer` [*options*]

Computes state space of domain/problem combination and computes goal distances for states. Write discovered states and goal distances to the file specified with -o. (The format written is suitable as input for the `-l` option of the corresponding filter tool.)

Return codes:

- 0 – successful traversal
- 1 – internal error
- 2 – traversal incomplete (unexpanded states in state table)

State space computation is only successful for more or less trivial domains: a depth-first search is performed on the state space graph whose edges are formed by the available actions. Search is curtailed by a depth limit (see `-d` option) and a limit to the maximum number of states (see `-n` option).

After traversal, the goal distances are computed for all states by running Dijkstra's algorithm on the transposed graph. States that are not reachable from the goal (e.g., all unexpanded states, as well as all deadlock states) will be assigned infinite distance.

In the current version of the CCBM system, action costs are assumed do be fixed (= 1). Future versions will allow to add a more elaborate cost models.

4.2.3 Filter

Synopsis:

problem`-filter` [*options*] <*observations-file* >*estimates-file*

For an action a and states x, x' such that $x' = a(x)$, let $\delta(x')$ be the goal distance of state x', and let $s(a)$ be the saliency value of a (see `:saliency` declaration). The probability of selecting a in state x is then defined by:

$$p(a \mid x) \propto \begin{cases} \gamma(x')s(a)(\beta + e^{\lambda\,\delta(x')}) & \text{if } x \models \mathit{pre}(a) \\ 0 & \text{otherwise} \end{cases}$$

β is the *bias*, see option `-b`, the value λ is the *weight factor*, see option `-f`. Note that in the presence of states with infinite goal distance, it is required that $\lambda \leq 0$. The factor $\gamma(x')$ is determined by by the history: If x' has *not* been visited before, $\gamma(x')$ is 1. Otherwise, it will be the value of the `-v`-option, which initially is 0. This leads the system to not re-enter states it has already visited.

Options

`-b` *bias* Set the value of β

`-f` *factor* Set the value of λ

`-l` *statefile* Load states and δ-values from *statefile*

`-r` Initialize random generator seed with current time

`-s` *smoothfile* Compute smoothed estimates and save them to *smoothfile*

`-t` *stepsize* Force (roughly) equidistant smoothing time intervals of size *stepsize* (a good value is 1). This makes plotting heatmaps much easier but takes longer time and wastes file space. (Normally, smoothing estimates are only computed for times where the state has changed.)

`-v` *factor* Set the γ-value for states already visited to *factor*. The default is 0. Setting the factor to 1 will effectively turn off history checking, values > 1 will invite the system to run in circles. Setting *factor* to values > 0 provides a (primitive) handle to capturing irrational behavior (as far as it expresses itself in redoing things).

Observation model flags

`USE_COMPRESSING`**:** The usage of different observation models, one for resampling, one for state exploration (as discussed in Sec. 6), can be enabled via the `USE_COMPRESSING` flag. An example is given in the `Examples/Weight` folder of the CCBM project.

`USE_ACTION_ESTIMATION`**:** Defining the `USE_ACTION_ESTIMATION` flag enables the estimation of the most likely action for each agent at each time step. An example can be seen in the `Examples` folder.

4.2.4 Validator

Synopsis:

problem`-validator` *plan-file {plan-file}*

Checks if the given plan(s) is/are valid and successful for the domain/problem combination. Each plan-file has to contain one plan, a sequence of execution events, where each line in a plan file is an event record in the following format:

$time$, $flag_1$, $action_1$, $flag_2$, $action_2$, ..., $flag_n$, $action_n$

$flag_i$ indicates if the action of agent has been newly started at $time$ ($flag_i = *$) or if it is continued from the last event record ($flag_i = empty$). $action_i$ indicates the action of agent i. For example:

```
0.0,*,(startEnter john),*,(startEnter paul),*,(startEnter george)
1.0,,(startEnter john),*,(finishEnter paul),*,(finishEnter george)
…
```

There are 3 possible results for the validator:

- `Success! Goalstate reached`: The plan was valid and successful
- `Success! Goalstate not reached`: The plan was valid and but did not reach the specified goal.
- `Failed to meet preconditons of action` a `(step:` t `slot:` i`)`: The plan is invalid. The action a at time t in agent slot i does not meet the preconditions.

5 Design Considerations

5.1 Translation Strategy

- Build type hierarchy from type definitions in domain and problem files.
- Compute Herbrand universe from collecting all constants defined in domain and problem files; compute the set of constants for each type.
- Compute ground operators by instantiating each operator scheme with each possible combination of parameters.

  Consider `(:type a b - foo c d - bar)` and

  (action ugh :parameters (?x - foo ?y - bar) :effect (baz ?x ?y)).

  This action will be converted into the ground actions

  `(:action ugh-a-c :effect (baz a c)`,

  `(:action ugh-a-d :effect (baz a d)`,

  `(:action ugh-b-c :effect (baz b c)`, and

  `(:action ugh-b-d :effect (baz b d)`.
- Normalize precondition- and effect formulae by removing quantifiers and converting quantifier-free formulae to disjunctive normal form (and simplifying the resulting formulae).
- Simplify operator set:
  - Remove operators impossible given initial state.

  - Remove operators irrelevant given goal state.
- Compute effective state bits.
- Compile particle filter tailored to effective operator set and state that utilizes
  - A global hash table for maintaining the set of already visited states
  - A per-particle hash table for maintaining the states visited by particle (required for γ-heuristics)
  - A per-particle history
- Compile state space explorer that
  - Use depth limited state space traversal in a forward run to compute reachable states and goal states.
  - Computes transposed incidence matrix and then uses Dijkstra's algorithm to compute the goal distances $\delta(x)$ for all reached states: all goal states are initialized with distance "0", all other states with distance "∞".

The heinous `forall`**.** Quantifier removal and translation into disjunctive normal form can lead to surprises (i.e., translation does not seem to terminate). Consider an expression such as

`(forall (?x - someType) (or (`$pred_1$ `?x) ...(`$pred_m$ `?x)))`

Now assume that the `someType` has n members. The resulting disjunctive normal form will then have m^n conjuncts. This may be surprisingly many. For instance, the following will create 2^{32} conjuncts, basically representing all possible values of a 32 bit word:

```
(:constants 0 1 2 3 .... 31 - bit)

(forall (?pos - bit)
        (or (bitSetAt ?pos)
            (not (bitSetAt ?pos))))
```

By double negation, you can have the same type of fun using the `exists` quantifier:

$$\forall\, x : p(x) == \neg\ \exists\, x : \neg\ p(x)$$

5.2 Action Selection

The action selection is governed by the density

$$p(a \mid x) \propto \gamma(x')s(a)(\beta + e^{\lambda\,\delta(x')})$$

where $x' = a(x)$. It is interesting to note, that this action selection model can be easily integrated onto a more general mathematical concept that will be made available at the implementation level in future versions of CCBM. If we have $\beta = 0$ and if we define

$$\begin{aligned} f_1(a, x) &:= \log \gamma(a(x)) \\ f_2(a, x) &:= \log s(a) \\ f_3(a, x) &:= \delta(a(x)) \end{aligned}$$

we can rewrite

$$\begin{aligned} p(a \mid x) &\propto \gamma(x')s(a) \exp\big(\lambda\,\delta(x')\big) \\ &= \exp(f_1(a, x)) \cdot \exp(f_2(a, x)) \cdot \exp(\lambda_3\, f_3(a, x)) \end{aligned}$$

where we have renamed λ to λ_3. Clearly,

$$= \exp\big(f_1(a, x) + f_2(a, x) + \lambda_3\, f_3(a, x)\big)$$

increasing flexibility by adding weights λ_1, λ_2 gives

$$\begin{aligned} &= \exp\big(\,\lambda_1 f_1(a, x) + \lambda_2 f_2(a, x) + \lambda_3 f_3(a, x)\big) \\ &= \exp\left(\sum_{k=1}^{3} \lambda_k f_k(a, x)\right) \end{aligned}$$

We thereby see that this definition of $p(a \mid x)$ is indeed equivalent to a Log-Linear model with *features* f_1, f_2, and f_3.

The functions $\gamma(x')$, $s(a)$, and $\delta(x')$ are heuristics for modeling how prone a person will be to select an action a in situation x. By recognizing that these heuristics can be embedded into a log-linear model, we also immediately see that we can flexibly add other heuristics (*e.g.*, specificity, ...), as long as this heuristic can be represented by some feature function $f_k(a, x)$.

We therefore arrive at the general model

$$p(a \mid x) \propto \exp\left(\sum_{k=1}^{n} \lambda_k f_k(a, x)\right)$$

with a flexibly extendable set of features f_k. In addition we might want to include a bias weight λ_0 together with a bias feature $f_0(a, x) := 1$, so that we can write

$$p(a \mid x) \propto \exp\left(\lambda_0 + \sum_{k=1}^{n} \lambda_k f_k(a, x)\right) = \exp\left(\sum_{k=0}^{n} \lambda_k f_k(a, x)\right)$$

If training data is available – a set of pairs $\{(a_1^*, x_1), (a_2^*, x_2), \ldots, (a_m^*, x_m)\}$ – we can estimate the parameter vector $\lambda = (\lambda_0, \ldots, \lambda_n)$ by computing (using numerical approximation) the maximum likelihood solution. This effectively allows to combine multiple heuristics and, specifically, to compute their relative weights by *training*.

This model translation is only possible if β is 0. Therefore, in order to remain compatible with future version soy CCBM, the use of the β-parameter is discouraged.

5.3 Multi-Agent Sampling: the `:agent` slot

In a single agent situation, it is straightforward to sample an action at time t by drawing

$$\tilde{a}_t \sim p(a \mid x_t)$$

using the equation

$$p(a \mid x) \propto \begin{cases} \gamma(x')s(a)(\beta + e^{\lambda\,\delta(x')}) & \text{if } x \models \mathit{pre}(a) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

In a multi-agent situation, each particle consists of *several* agents that are executing their individual actions in parallel. If there are m agents, the compound action a_t is an m-tuple $a_t^{(1:m)}$ that contains an individual action for every agent. We have to clarify two aspects: (i) the semantics of an action tuple $a_t^{(1:m)}$ and (ii) the density $p(a_t^{(1:m)} \mid x_t)$ from which we need to sample such an action tuple.

Let $a \equiv a^{(1:m)}$ be an action tuple for an agent family of size m. One option for defining precondition and effect of a would be

$$\mathit{pre}(a) := \bigwedge_{i \in 1:m} \mathit{pre}(a_i) \tag{2}$$

$$\mathit{eff}(a) := \bigwedge_{i \in 1:m} \mathit{eff}(a_i). \tag{3}$$

from which the situation calculus semantics in terms of Do follow immediately. This *parallel* semantics has the disadvantage of creating the usual surprises with operators like

```
(action enter-exclusive
        :parameters (?room - room ?person - person)
        :agent ?person
        :precondition (empty ?room)
        :effect (and (not (empty ?room))
                     (occupied-by ?room ?person)))
```

that arise as soon as there are two or more persons. Therefore, we regard actions as *transactions* for which we require *serializability*. We therefore define the semantics of a compound action $a \equiv a^{(1:m)}$ by

$$Do(a) := Do(a^{(m)}) \circ \cdots \circ Do(a^{(1)}). \tag{4}$$

The next question is how to sample a compound action.

$$\tilde{a}_t^{(1:m)} \sim p(a_t^{(1:m)} \mid x_t) \tag{5}$$

Let $\alpha_t^{(1:m)} = \big(ag(a_t^{(1)}), \ldots, ag(a_t^{(m)})\big)$ be the tuple of execution agents associated with the tuple of component actions. It seems reasonable to assume that at every time t each agent can select exactly one action from its executable actions. Therefore, it must be that every agent name $\alpha \in 1{:}m$ occurs exactly once in the tuple $\alpha_t^{(1:m)}$. In other words, $\alpha_t^{(1:m)}$ must be some permutation of the set of agent names.

In case there are no executable actions for agent $\alpha_t^{(i)}$, the agent is *blocked* and does nothing (this for instance can happen if one agent is waiting for other agents to finish their activities)[1]. So the compound actions $a_t^{(i)}$ are either elements of the action alphabet A or the blocked action *blocked*, for which $Do(blocked) = id$.

Let x_t be a state and $\alpha_t^{(1:i-1)}$ be a tuple of agent names. The probability of selecting some component action $a_t^{(i)}$ with associated execution agent $\alpha_t^{(i)} = ag(a_t^{(i)})$ given that the agents $\alpha_t^{(1:i-1)}$ are not available can now be defined by:

$$p(a_t^{(i)} \mid \alpha_t^{(1:i-1)}, x_t) = \begin{cases} 1 & \text{if } A_t(\alpha_t^{(1:i-1)}, x_t) = \varnothing \wedge a_t^{(i)} = blocked \\ p(a_t^{(i)} \mid x_t) & \text{if } a_t^{(i)} \in A_t(\alpha_t^{(1:i-1)}, x_t) \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where $A_t(\alpha_t^{(1:j)}, x_t)$ is the effective action alphabet defined by

$$A_t(\alpha_t^{(1:j)}, x_t) = \{\, a \in A \mid (x_t \models pre(a)) \wedge (ag(a) \notin \alpha_t^{(1:j)}) \,\} \tag{7}$$

This means, for selecting an action we consider those actions whose preconditions are met by the state and whose agents are still available. If such actions exist,

[1] If all agents are blocked, a deadlock has occurred.

the probability of selecting one of them is given by (1), otherwise the agent will deterministically be blocked.

In order to sample from $p(a_t^{(1:m)} \mid x_t)$ we first observe that by the chain rule

$$p(a_t^{(1:m)} \mid x_t) = \prod_{i=1}^{m} p(a_t^{(i)} \mid a_t^{(1:i-1)}, x_t) \tag{8}$$

If we then employ (6) and define

$$p(a_t^{(i)} \mid a_t^{(1:i-1)}, x_t) := p(a_t^{(i)} \mid \alpha_t^{(1:i-1)}, x_t') \tag{9}$$

where we use

$$x_t' := \big(Do(a_t^{i-1}) \circ \cdots \circ Do(a_t^{(1)})\big)\, x_t \text{ and} \tag{10}$$

$$\alpha_t^{(1:i-1)} := \big(ag(a_t^{(1)}), \ldots, ag(a_t^{(i-1)})\big) \tag{11}$$

we have a straightforward mechanism for sampling composite actions. Note that the tuple $\alpha_t^{(1:m)}$ defines the specific serialization schedule for the composite actions sampled this way. The sampling strategy outlined here ensures that $\alpha_t^{(1:m)}$ is a random sample from the permutations of the set of agent names, where, given the $p(a_t^{(i)} \mid x_t)$ are of comparable magnitude, agents with more available actions will tend to occur earlier in the tuple. A different strategy would be to *first* draw a random permutation $\alpha_t^{(1:m)}$ and then draw actions according to this permutation. We don't know whether this would make any significant difference.

Note: The current implementation uses a simpler strategy where $\alpha_t^{1:m}$ is restricted to be the exact sequence $1{:}m$. (So for the first component action $a_t^{(1)}$ only agent 1 is considered, for the second component action $a_t^{(2)}$ only agent 2, and so on.) This is basically not correct: consider a situation where agent 1 has only one action a available that will always block agent 2 (and possibly vice versa). Here, always the compound action $(a, \mathit{blocked})$ will be sampled. although in reality (and in the algorithm above) both $(a, \mathit{blocked})$ and $(\mathit{blocked}, a)$ will occur.

5.4 Simple Durative Actions: the `:duration` slot

The effects of actions in CCBM are executed immediately. That is, if a_t is the action selected for execution in state x_t, its effects will be visible immediately at the next time step's state $x_{t+\Delta_t}$, where Δ_t is the smallest time step (to keep discussions simple, we will simply consider $\Delta_t = 1$). Thus, all actions in CCBM have the same duration Δ_t. But consider a real world action a such as moving from location u to

location v. Such an action will not happen instantaneously. Rather, an agent will start moving from its origin u at some time t and it will arrive at its destination v at some time $t + \delta_t$, where typically $\delta_t > \Delta_t$ – the time required for performing the action a – is governed by some density $p(\delta_t \mid a, x_t)$.

PDDL provides the mechanism of durative actions, however without explicit consideration of probabilistic action durations. For this reason, we have used a somewhat simplified mechanism in CCBM, where *every* action can be given a probabilistic duration. A conventional PDDL durative action would then be represented by a pair of actions (a_{start}, a_{stop}), where a_{start} records the fact that a is ongoing in the state. a_{stop} establishes the final outcome of the action.

```
(action start-move
        :parameters (?from ?to - location ?p - person)
        :agent ?p
        :duration (exponential (distance ?from ?to))
        :precondition (not (is-moving ?p)
                           (at ?p ?from))
        :effect (and (is-moving ?p)
                     (is-moving-from-to ?p ?from ?to)))
(action stop-move
        :parameters (?from ?to - location ?p - person)
        :agent ?p
        :precondition (is-moving-from-to ?p ?from ?to)
        :effect (and (at ?p ?to)
                     (not (is-moving ?p))
                     (not (is-moving-from-to ?p ?from ?to))))
```

In this example, `is-moving` is the action lock, while `is-moving-from-to` is the state record. Besides managing the interplay between start and stop actions, action lock and state record could also be used to enforce invariants that need to hold while the action is ongoing:

```
(action untie-shoe
        :parameters (?p - person)
        :agent ?p
        :precondition (and (wears-shoes ?p)
                           (not (is-moving ?p)))
        ...)
```

Then modeling a durative action by a pair (a_{start}, a_{stop}), a mechanism is required for specifying the time interval δ_t that will elapse between executing a_{start} and a_{stop}.

Once an agent has selected a_{start} at time t, it is then idling until a time interval of time $\tilde{\delta}_t \sim p(\delta_t \mid a_{start}, x_t)$ has passed before then executing the next action

The `:duration`-slot is used to specify the density $p(\delta_t \mid a_{start}, x_t)$ from which the duration δ_t is sampled. In CCBM a simplified model $p(\delta_t \mid a_{start})$ is used that does not allow any dependencies on the state.

The concrete mechanism for sampling from $p(\delta_t \mid a_{start})$ uses the following technique: Let $F(\delta_t)$ be the cdf of $p(\delta_t \mid a_{start})$. At every step from t to $t + \Delta_t$ an agent that has chosen a_{start} at time $t_0 \leq t$ will stop waiting with probability

$$\frac{F((t - t_0) + \Delta_t) - F(t - t_0)}{1 - F(t - t_0)}$$

The resulting waiting time δ_t will then be a sample from $p(\delta_t \mid a_{start})$.

6 Heavy Tails

A particle filter will usually operate with resampling engaged. When filtering long observation sequences, it is in this case very important to heed the advice of using heavy tailed observation densities[2] – that is distributions that allow arbitrary outliers without penalizing them with extremely low probabilities, such as the Cauchy distribution. In general, we label distributions as "heavy tailed" that assign a surprisingly high probability to the "impossible" case. "Heavy tailed" in this situation can be quite pronounced: there have been experiments with location sensors that required the probability for a person being *outside* the sensor region in case the sensor says the person is *inside* to be just 2% lower than the probability for the person really being inside. Now, this is clearly a heavy tail. Of course, the real error ratio of the sensor (person being outside when sensor says inside) is something in the order of 1 : 1000, not 1 : 1.02.

To understand this phenomenon (and its implications), we will analyze this in a simple artificial setting.

Consider a simple Markov Model with state space $X = \{0, 1, 2\}$ and observation space $Y = \{S, A, B\}$, where the initial state is 0 and the transition probabilities are defined as in Fig. 2. An observation model is given by

[2]The concept of "heavy Tails" has a rather precise definition in probability theory, and the densities we're using as examples here do not really meet this definition …

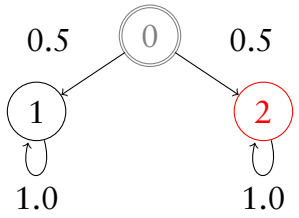

Figure 2: Example model

$$p(y = S \mid x = 0) = 1 \tag{12}$$

$$p(y \mid x = 1) = \begin{cases} 0.9 & \text{if } y = A \\ 0.1 & \text{if } y = B \end{cases} \tag{13}$$

$$p(y \mid x = 2) = \begin{cases} 0.1 & \text{if } y = A \\ 0.9 & \text{if } y = B \end{cases} \tag{14}$$

(we refer to this model as "observation model 1", OM1). An example observation sequence is

$$y_{1:21} = \langle S, \underbrace{A, \ldots, A}_{10 \text{ times}}, \underbrace{B, \ldots, B}_{10 \text{ times}} \rangle \tag{15}$$

The joint densities $p(x_{1:t}, y_{1:t})$ and conditional densities $p(x_{1:t} \mid y_{1:t})$ can then easily be computed by the forward filtering equations for hidden Markov models, giving the results shown in Fig. 3. Note that, given this observation sequence, first state 1 becomes increasingly more probable until time $t = 11$, whereafter the B observations "revive" state 2, finally leaving both states at the same probability of 0.5.

Trying to compute the same quantities using a particle filter with $N = 10$ particles (where 5 initially choose state 1, and 5 choose state 2), gives – for this simple model – the same values, as shown in Fig. 4, left.

As can be seen in Fig. 4, right, the effective particle count quickly drops to about 5. If we now decide to resample at time $t = 11$ in order to restore the particle count, we get the run shown in Fig. 5. In Fig. 5, right, we find that no particle in state 2 survives the resampling stage. This is no surprise, since at time $t = 11$ the probability of state 2 has dropped below 10^{-9}, while state 1 has probability of almost 1. It would require about 10^9 particles to have a decent chance that at least one state 2 particle survives resampling.

The result regarding the final state estimate is disastrous: State 2 has probability 0, while state 1 has probability 1. This is way off the exact result – note, however,

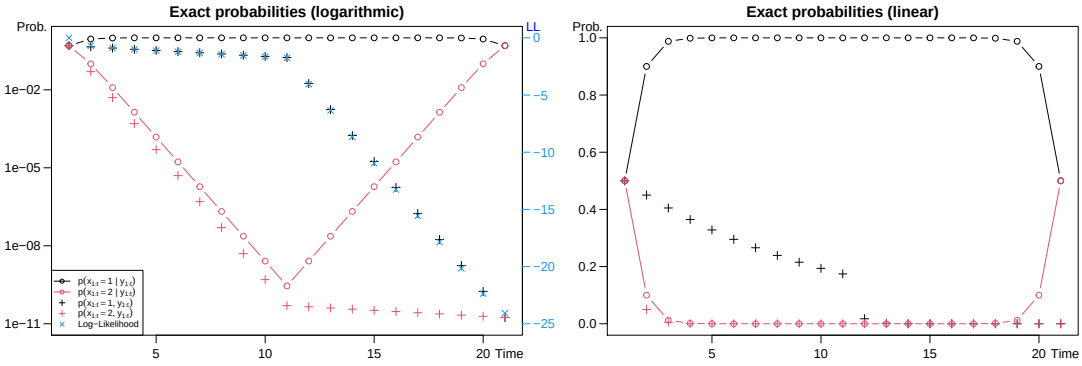

Figure 3: Exact filtering with OM1

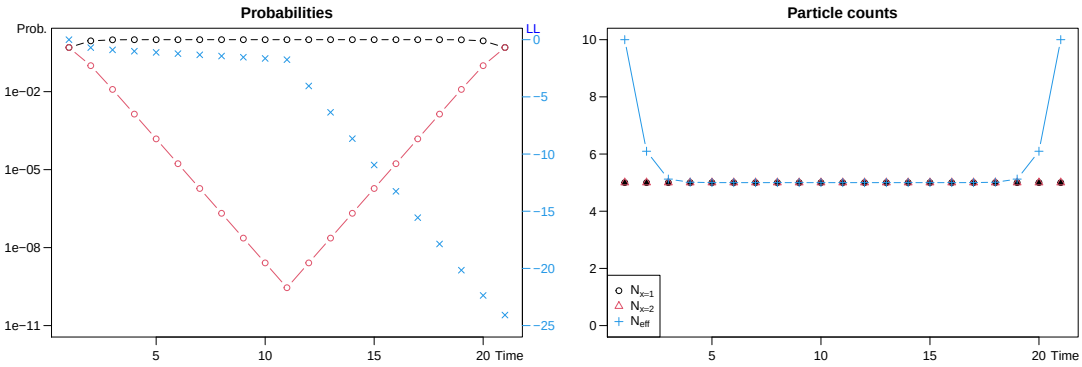

Figure 4: Particle filtering with OM1, without resampling

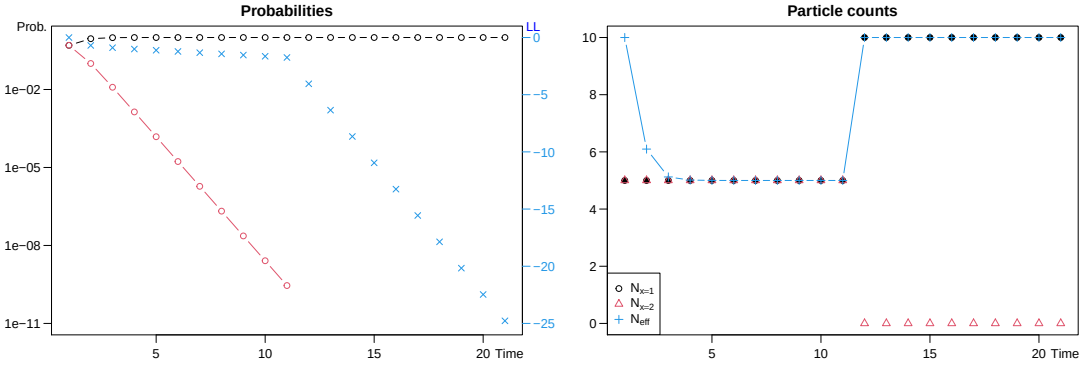

Figure 5: Particle filtering with OM1, with resampling

that the impact of this disaster on the Log-Likelihood is not very high. Which, on the other hand, means that Log-Likelihood is not always a good measure to compare the quality of different particle filter runs.

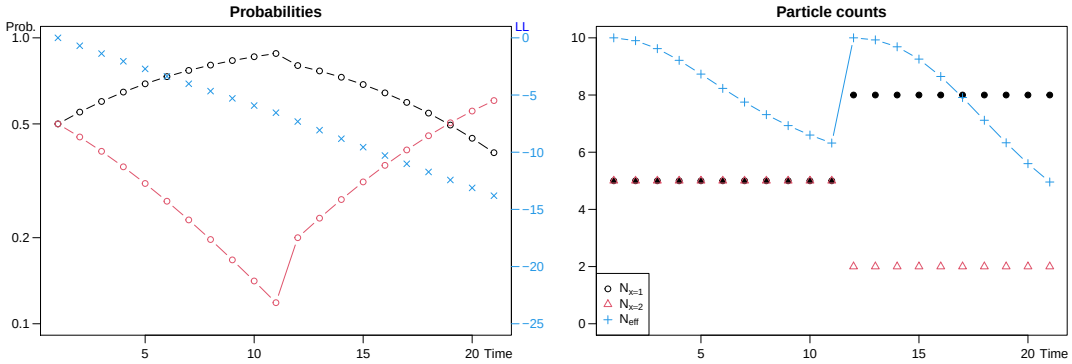

Figure 6: Particle filtering with OM2, with resampling

What can be done to solve this problem? – One option is to make the observations much more heavy-tailed. In other words: shift the ratio of $p(Y = y \mid X)/p(Y \neq y \mid X)$ closer to 1. In OM1, this ratio is about $0.9/0.1 = 9$. If we instead use the following model OM2, defined by:

$$p(y \mid x = 1) = \begin{cases} 0.55 & \text{if } y = A \\ 0.45 & \text{if } y = B \end{cases} \tag{16}$$

$$p(y \mid x = 2) = \begin{cases} 0.45 & \text{if } y = A \\ 0.55 & \text{if } y = B \end{cases} \tag{17}$$

We get a ratio of $0.55/0.45 = 1.22$. With this model, a particle filter produces the results given in Fig. 6. Regarding the final probabilities of the states 1 and 2 this is much better! The reason is clear: as an observation can change the relative probability of particles by only $\approx 20\%$, even then "weaker" particles have a chance to survive resampling. (If resampling at the same point in time.)

Unfortunately, this strategy of replacing the true observation densities by much more heavy-tailed versions (i.e., by much less informative versions) has the significant disadvantage of changing the likelihood and creating "wrong" probability values when looking at them from the viewpoint of computing expectations and estimating parameters.

However one can use the particle state space from a heavy tailed run by then doing a *second* run using the true densities *without* resampling. This gives the result shown in Fig. 7 Here we do get the true values for probabilities and densities, *as well as* a distribution of particle numbers to states that correctly reflects the distribution of particle states during the filtering process.

Note that this can also be done online:

- Resample according to weights computed using permissive (heavy-tailed)

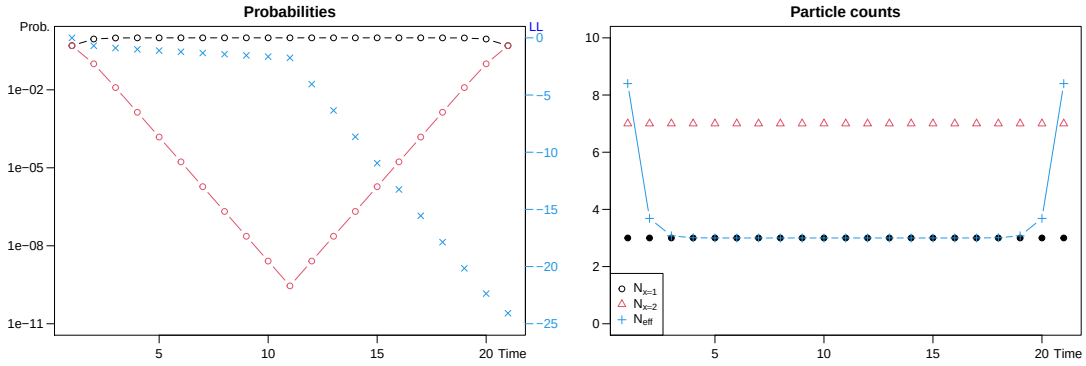

Figure 7: Particle filtering with OM2, using a secondary run with OM1

.

densities

- Compute probabilities and likelihoods using the true weights.

So the general idea is simply: keep the state space sufficiently populated with particles using permissive densities, but use true densities for computing probabilities *given* that state space sampling.

An even more drastic example is the one created by using the following observation model OM3:

$$p(y = S \mid x = 0) = 1 \tag{18}$$

$$p(y \mid x = 1) = \begin{cases} 0.9 & \text{if } y = A \\ 0.1 & \text{if } y = B \end{cases} \tag{19}$$

$$p(y \mid x = 2) = \begin{cases} 0.01 & \text{if } y = A \\ 0.99 & \text{if } y = B \end{cases} \tag{20}$$

In the exact analysis we get the results in Fig. 8, which show that, while first state 1 looks most promising, in reality state 2 is the most probable state, after the sequence of B observations has been processed. Now, if this is handled by the particle filter with resampling, this filter will kill all state 2 particles at time $t = 11$, giving results that are completely wrong, as can be seen in Fig. 9. But, if we again use OM2 for a filter run and a re-run with the true densities afterwards, we arrive at a true estimate Fig. 10.

Note that here we have successfully used the *same* state space sampling – computed using OM3 – for two quite different observation models. This means that one can *precompute* a state space sampling with some suitably permissive observation density and then try *different* "tight" densities in order to find the

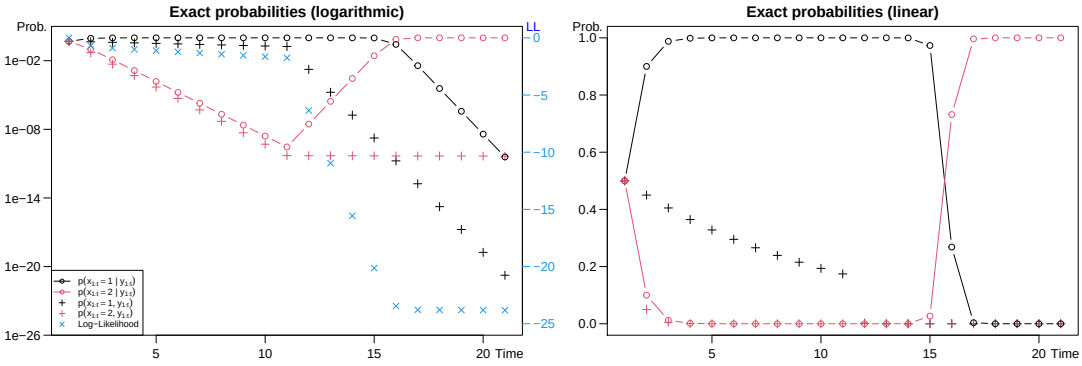

Figure 8: Exact filtering with OM3

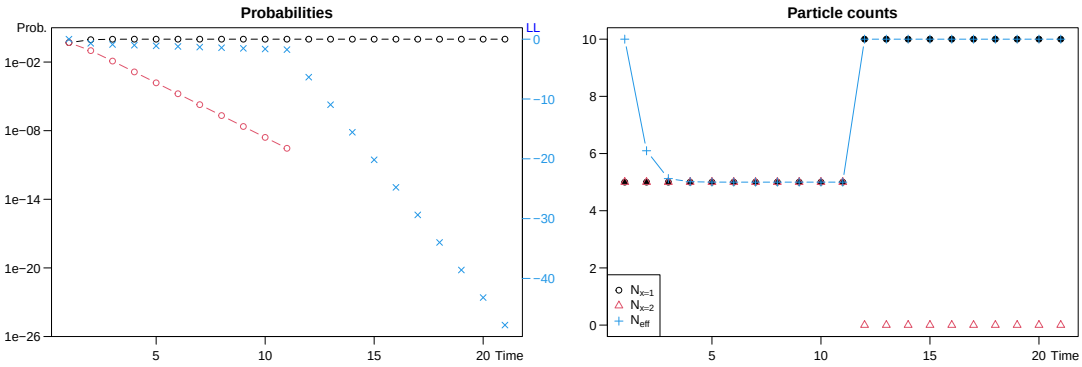

Figure 9: Particle filtering with OM3, with resampling. Note the different LL-scale in the left plot.

.

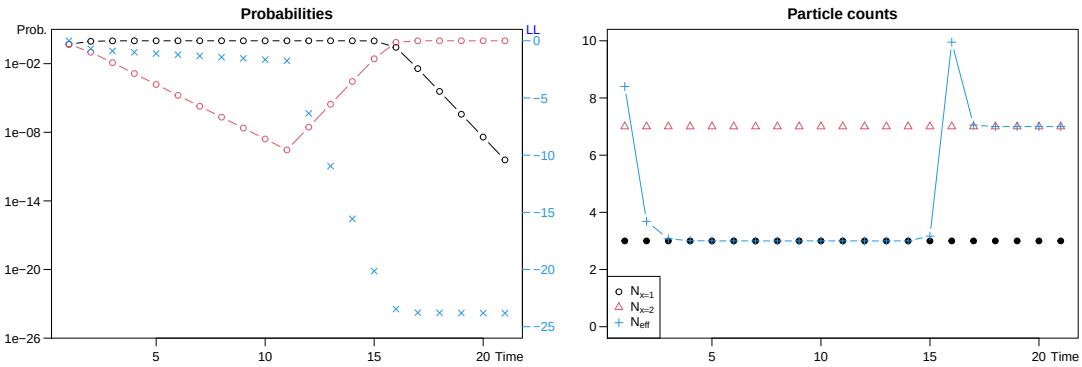

Figure 10: Particle filtering with OM2, using a secondary run with OM3

.

"best" one. The extreme case – running the particle filter without observation in simulation mode and then just weighting the simulation runs using the observation model – is equivalent to running the filter without resampling at all; this usually only makes sense for small state spaces and very good models. Fig. 4 is an example of this extreme approach.

Note also that in some settings, rather than using loose observations you also could try very low $(0 + \epsilon)$ resampling thresholds, based on the rationale that tight observations result in fast drop of particle counts.

Morale: The interaction between "looseness" of observations and resampling threshold level is not always straightforward to understand and problem dependent. Good settings need to be identified experimentally, hopefully leading some day to insights on when to use what settings.

6.1 Compressing Probabilities

Let $p_X(\cdot)$ be the density function of some random variable X and let $a, b \in X, a \neq b$ be two realizations of X, for which we have the density ratio $p_X(a)/p_X(b) = c_X$. We now would like to introduce another density $p_Y(\cdot)$ where the corresponding density ratio $p_Y(a)/p_Y(b) = c_Y$ yields a value closer to 1, such that either $1 \leq c_Y \leq c_X$ or $0 < c_X \leq c_Y \leq 1$.

For instance, for the scenario above we changed from $c_X = 0.9/0.1 = 9$ to $c_Y = 0.55/0.45 = 1.22$.

In other words, we require that $\mid \ln c_Y \mid < \mid \ln c_X \mid$. One straightforward option for achieving this is by requiring

$$p_Y(x) \propto \big(p_X(x)\big)^\eta$$

for some $0 \leq \eta < 1$. From this we then have:

$$\begin{aligned}
\mid \ln c_Y \mid &= \mid \ln \frac{p_Y(a)}{p_Y(b)} \mid \\
&= \mid \ln \frac{\alpha(p_X(a))^\eta}{\alpha(p_X(b))^\eta} \mid \\
&= \mid \ln \left(\frac{p_X(a)}{p_X(b)} \right)^\eta \mid \\
&= \eta \mid \ln c_X \mid < \mid \ln c_X \mid
\end{aligned}$$

By the fact that $c_Y = c_X^\eta$, we can compute the required η by $\eta = \ln c_Y / \ln c_X$. For the above example we get $\eta = \ln 1.22 / \ln 9 = 0.0913\ldots$

In case $p_X(x) = \mathcal{N}(x;\ \mu, \sigma)$ is a normal density, we get the result:

$$\begin{aligned} p_Y(x) &= \alpha\big(p_X(x)\big)^\eta \\ &= \alpha\left(\frac{1}{\sqrt{2\pi}\sigma}\exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right\}\right)^\eta \\ &= \left(\alpha\left(\frac{1}{\sqrt{2\pi}\sigma}\right)^\eta\right)\left(\exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right\}\right)^\eta \end{aligned}$$

using $\alpha' := \alpha(1/(\sqrt{2\pi}\sigma))^\eta$

$$= \alpha' \exp\left\{-\frac{\eta}{2}\frac{(x-\mu)^2}{\sigma^2}\right\}$$

using $\sigma' := \sigma/\sqrt{\eta}$

$$= \alpha' \exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma'^2}\right\}$$

so we must have $\alpha' = 1/(\sqrt{2\pi}\sigma')$, thus

$$\begin{aligned} &= \frac{1}{\sqrt{2\pi}\sigma'}\exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma'^2}\right\} \\ &= \mathcal{N}(x \mid \mu, \sigma') = \mathcal{N}(x \mid \mu, \sigma/\sqrt{\eta}) \end{aligned}$$

This completely explains why just increasing the variance of observation models using normal densities will magically improve the performance of particle filtering, and why this strategy is in fact reasonable.

6.2 Resampling Using a Heavy-Tailed Density

It is possible to maintain two sets of weights in a particle filter, a set $r_t^{(i)}$ computed from liberal (*e.g.*, "compressed") densities that is used for populating the state space and a set $w_t^{(i)}$ which contains the weights for representing the true density and that can be used for computing the true likelihood.

Let $q(x_1)$, $q(x_t \mid x_{t-1})$, and $q(y_t \mid x_t)$ be the "compressed" densities, while $p(x_1)$, $p(x_t \mid x_{t-1})$, and $p(y_t \mid x_t)$ are the true densities.

- Sampling the system model

– At start $t = 1$:

$$
\begin{aligned}
x_1^{(i)} &\sim q(x_1) \\
w_{1|0}^{(i)} &= \frac{p(x_1^{(i)})}{q(x_1^{(i)})} \\
r_0^{(i)} &= N^{-1} \\
w_0^{(i)} &= N^{-1}
\end{aligned}
$$

– At time step $t - 1 \to t$

$$
\begin{aligned}
x_t^{(i)} &\sim q(x_t \mid x_{t-1}^{(i)}) \\
w_{t|t-1}^{(i)} &= \frac{p(x_t^{(i)} \mid x_{t-1}^{(i)})}{q(x_t^{(i)} \mid x_{t-1}^{(i)})}
\end{aligned}
$$

- Incorporating observations y_t

$$
\begin{aligned}
\tilde{w}_t^{(i)} &= w_{t-1}^{(i)} w_{t|t-1}^{(i)} p(y_t \mid x_t^{(i)}) & \qquad \tilde{r}_t^{(i)} &= r_{t-1}^{(i)} q(y_t \mid x_t^{(i)}) \\
c_t^w &= \sum_{i=1}^{N} \tilde{w}_t^{(i)} & \qquad c_t^r &= \sum_{i=1}^{N} \tilde{r}_t^{(i)} \\
w_t^{(i)} &= \frac{1}{c_t^w} \tilde{w}_t^{(i)} & \qquad r_t^{(i)} &= \frac{1}{c_t^r} \tilde{r}_t^{(i)}
\end{aligned}
$$

Where c_t^w ensures that the $w_t^{(i)}$ sum to one – it is the likelihood $p(y_t \mid y_{1:t-1})$. Likewise, $c_t^r = q(y_t \mid y_{1:t-1})$,

- Resampling

For $j \in 1 : N$

$$
\begin{aligned}
i &\sim p(i) = r_t^{(i)} \text{ Draw particle indices proportional to } r\text{-weights} \\
x_t'^{(j)} &= x_t^{(i)} \\
w_t'^{(j)} &\propto \frac{w_t^{(i)}}{r_t^{(i)}} \\
r_t'^{(j)} &= N^{-1}
\end{aligned}
$$

Proposition 1. *After resampling:*

$$w_t'^{(j)} \propto \frac{p(x_{1:t}^{(i)} \mid y_{1:t})}{q(x_{1:t}^{(i)} \mid y_{1:t})}$$

Proof. We use a proof by induction.

(*inductive step*) For resampling at time t: Induction assumption is that at some time $t_r < t$ we had a prior resampling and the claim holds for all resampling times $< t$, such that, specifically

$$w_{t_r}^{(i)} = \frac{p(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})}{q(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})}$$

- First, note that by construction of the weights we have:

$$w_t^{(i)} = w_{t_r}^{(i)} \prod_{s=t_r+1}^{t} \left(\frac{1}{c_s^w} w_{s|s-1} p(y_s \mid x_s^{(i)})\right) \tag{21}$$

$$= w_{t_r}^{(i)} \prod_{s=t_r+1}^{t} \left(\frac{1}{p(y_s \mid y_{1:s-1})} \frac{p(x_s^{(i)} \mid x_{s-1}^{(i)})}{q(x_s^{(i)} \mid x_{s-1}^{(i)})} p(y_s \mid x_s^{(i)})\right) \tag{22}$$

Simplifying the product using the independcies

$$= w_{t_r}^{(i)} \frac{p(y_{t_r+1:t} \mid x_{t_r+1:t}^{(i)}) p(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)})}{p(y_{t_r+1:t} \mid y_{1:t_r})} \frac{1}{q(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)})} \tag{23}$$

By assumption $w_{t_r}^{(i)}$ is some $w_{t_r}'^{(j)}$ computed by the resampling step above (this is a bit sloppy, as we're not clearly distinguishing between i and j index. But since j is a copy of i anyway, this really doesn't matter)

$$= \frac{p(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})}{q(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})} \frac{p(y_{t_r+1:t} \mid x_{t_r+1:t}^{(i)}) p(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)})}{p(y_{t_r+1:t} \mid y_{1:t_r})} \frac{1}{q(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)})} \tag{24}$$

Shuffling factors

$$= \frac{p(y_{t_r+1:t} \mid x_{t_r+1:t}^{(i)}) p(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)}) p(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})}{p(y_{t_r+1:t} \mid y_{1:t_r})} \frac{1}{q(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)}) q(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})} \tag{25}$$

By independence

$$= \frac{p(y_{t_r+1:t} \mid x_{1:t}^{(i)}, y_{1:t_r}) p(x_{t_r+1:t}^{(i)} \mid x_{1:t_r}^{(i)}, y_{1:t_r}) p(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})}{p(y_{t_r+1:t} \mid y_{1:t_r})} \times \frac{1}{q(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)}) q(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})} \tag{26}$$

By the chain rule

$$= \frac{p(y_{t_r+1:t} \mid x_{1:t}^{(i)}, y_{1:t_r}) p(x_{1:t}^{(i)} \mid y_{1:t_r})}{p(y_{t_r+1:t} \mid y_{1:t_r})} \frac{1}{q(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)}) q(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})} \tag{27}$$

By Bayes rule

$$= p(x_{1:t}^{(i)} \mid y_{1:t}) \frac{1}{q(x_{t_r+1:t}^{(i)} \mid x_{t_r}^{(i)}) q(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})} \tag{28}$$

Simplifying the right factor starts by exploiting independence

$$= p(x_{1:t}^{(i)} \mid y_{1:t}) \frac{1}{q(x_{t_r+1:t}^{(i)} \mid x_{1:t_r}^{(i)}, y_{1:t_r}) q(x_{1:t_r}^{(i)} \mid y_{1:t_r}^{(i)})} \tag{29}$$

And then using the chain rule

$$= p(x_{1:t}^{(i)} \mid y_{1:t}) \frac{1}{q(x_{1:t}^{(i)} \mid y_{1:t_r})} \tag{30}$$

Which by itself is a reassuring result, as it exactly states: the samples $x_{1:t}^{(i)}$ have been sampled from the density $q(x_{1:t}^{(i)} \mid y_{1:t_r})$, which is indeed true. (The resampling step at time t_r has put the observations $y_{1:t_r}$ into the sampling distribution; afterwards particle states have been just simulated from the system model q.)

- For the r-weights, we get:

$$r_t^{(i)} = r_{t_r}^{(i)} \prod_{s=t_r+1}^{t} \frac{q(y_s \mid x_s^{(i)})}{q(y_s \mid y_{1:s-1})} \tag{31}$$

since $r_{t_r}^{(i)} = N^{-1}$

$$\propto \prod_{s=t_r+1}^{t} \frac{q(y_s \mid x_s^{(i)})}{q(y_s \mid y_{1:s-1})} \tag{32}$$

expanding the factors

$$= \frac{q(y_{t_r+1:t} \mid x_{t_r+1:t})}{q(y_{t_r+1:t} \mid y_{1:t_r})} \tag{33}$$

and also exploiting some independencies

$$= \frac{q(y_{t_r+1:t} \mid x_{1:t}, y_{1:t_r})}{q(y_{t_r+1:t} \mid y_{1:t_r})} \tag{34}$$

- We then look at the value of $w_t'^{(j)}$. By definition

$$w_t'^{(j)} = \frac{w_t^{(i)}}{r_t^{(i)}} \tag{35}$$

by the above results

$$\propto p(x_{1:t}^{(i)} \mid y_{1:t}) \frac{1}{q(x_{1:t}^{(i)} \mid y_{1:t_r})} \frac{q(y_{t_r+1:t} \mid y_{1:t_r})}{q(y_{t_r+1:t} \mid x_{1:t}, y_{1:t_r})} \tag{36}$$

Immediately, by Bayes rule

$$= \frac{p(x_{1:t}^{(i)} \mid y_{1:t})}{q(x_{1:t}^{(i)} \mid y_{1:t})} \tag{37}$$

(*base case*) At time t, we have the first resampling. Following the development above, we get the product expansion:

$$w_t^{(i)} = N^{-1} \prod_{s=1}^{t} \Big(\frac{1}{p(y_s \mid y_{1:s-1})} \frac{p(x_s^{(i)} \mid x_{s-1}^{(i)})}{q(x_s^{(i)} \mid x_{s-1}^{(i)})} p(y_s \mid x_s^{(i)}) \Big) \tag{38}$$

$$\propto \frac{p(y_{1:t} \mid x_{1:t}^{(i)}) p(x_{1:t}^{(i)})}{p(y_{1:t})} \frac{1}{q(x_{1:t}^{(i)})} \tag{39}$$

where we have used identities such as $p(x_1 \mid x_0) \equiv p(x_1)$ and $p(y_1 \mid y_{1:0}) \equiv p(y_1)$. For the r-weights we likewise get

$$r_t^{(i)} = N^{-1} \prod_{s=1}^{t} \frac{q(y_s \mid x_s^{(i)})}{q(y_s \mid y_{1:s-1})} \tag{40}$$

$$\propto \frac{q(y_{1:t} \mid x_{1:t})}{q(y_{1:t})} \tag{41}$$

so, it easily follows via Bayes rule that

$$w_t'^{(j)} = \frac{w_t^{(i)}}{r_t^{(i)}} \propto \frac{p(x_{1:t}^{(i)} \mid y_{1:t})}{q(x_{1:t}^{(i)} \mid y_{1:t})} \tag{42}$$

□

This shows that the the particle weights are indeed meaningful importance sampling values.

On the one hand, this result is not really surprising or novel: It simply states that, of course, with importance sampling, you can use just some other density q for approximating a density p. On the other hand it opens up additional degrees of freedom regarding the definition of the "inferential" model p and the "explorative" model q.

If we can use *two* sets of weights – wouldn't we also be able to use *arbitrary* numbers of weight sets? (i) this should be straightforward to do (ii) but why should we do this?? – Because we get a kind of Bayesian inference for free: let $p_k(x' \mid x)$ and $p(p_k \mid y)x$ be the densities associated with the k-th weight set ${}_k w_t^{(i)}$. Let $p(\theta)$ be the prior density of the parameter distribution, let $\{\theta_k\}_{k=1}^K, \theta_k \sim p(\theta)$ be a set of samples from the parameter density and define $p_k(x' \mid x) := p(x' \mid x; \theta_k)$.

We get:

$$p(\theta \mid y_{1:T}) = Bayes \ldots \tag{43}$$

$$= \alpha^{-1} \sum_{k=1}^{K} p_k(y_{1:T}) \delta(\theta - \theta_k) \tag{44}$$

Where $\alpha = \sum_{k=1}^K p_k(y_{1:T})$

6.3 Effect of Heavy-Tailed Resampling

When using root-based "compression" (Sec. 6.1), the divergence of weights is slowed down, but there is no fundamental change in the shape of the density function. From one perspective, this will have simply the effect that N_{eff} will stay longer above the resampling threshold – thus allowing more observations to accumulate in the particle population, before particles are removed. Or, vice versa, a higher resampling threshold can be chosen so that resampling occurs before particle weights have fatally diverged.

But we can also completely change the functional shape of the q-density, for instance by defining:

$$q(x) \propto \sigma(\kappa p(x)) - \frac{1}{2},$$

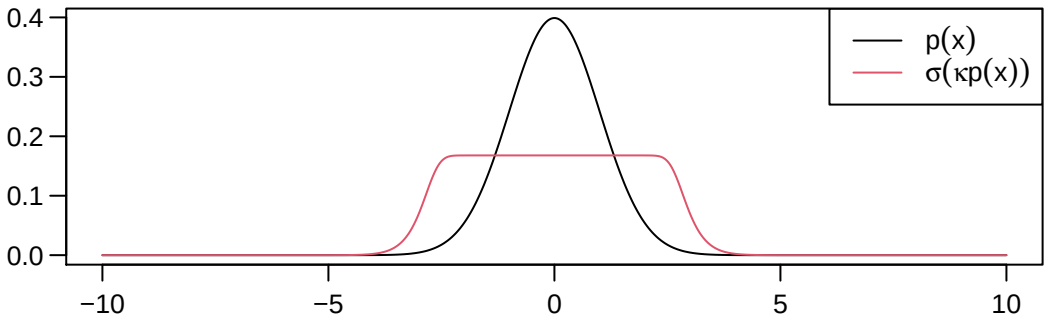

Figure 11: Using a sigmoid function to distort a density

.

where σ is the sigmoid function. This would have the effect to gradually transform the original density $p(x)$ to "uniform" density, cf. Fig. 11. By this we can allow particles to move freely inside the "plateau" region, without forcing them to the mean of the observations.