# Data-driven Sequential Monte Carlo in Probabilistic Programming

**Yura Perov** **Tuan Anh Le** **Frank Wood**
Department of Engineering Science, University of Oxford
`{perov,tuananh,fwood}@robots.ox.ac.uk`

## Abstract

Most of Markov Chain Monte Carlo (MCMC) and sequential Monte Carlo (SMC) algorithms in existing probabilistic programming systems suboptimally use only model priors as proposal distributions. In this work, we describe an approach for training a discriminative model, namely a neural network, in order to approximate the optimal proposal by using posterior estimates from previous runs of inference. We show an example that incorporates a data-driven proposal for use in a non-parametric model in the Anglican probabilistic programming system [9]. Our results show that data-driven proposals can significantly improve inference performance so that considerably fewer particles are necessary to perform a good posterior estimation.

## 1 Background

We consider a generative model $p(x_{1:N}, y_{1:N})$ with hidden variables $x_{1:N}$ and observations $y_{1:N}$. In probabilistic programming, we let the observing random variable $y_n$ be the value of the $n$th observe, and the hidden variables $\mathbf{x}_n = x_{1:n}$ be the execution trace before this observe. The goal of SMC inference in probabilistic programming is to sample from a family of distributions $p(\mathbf{x}_{1:n}|y_{1:n})$ for $n = 1, \ldots, N$. This is achieved by generating a set of particles $\{\mathbf{x}_{1:n}^{(p)}\}_{p=1}^{P}$ and the corresponding importance weights $\{w_n^{(p)}\}_{p=1}^{P}$. We approximate the target distribution as $\sum_{p=1}^{P} w_n^{(p)} \delta_{\mathbf{x}_{1:n}^{(p)}}(\mathbf{x}_{1:n})$. Particles at time $n$ are generated using a chosen proposal distribution $q_n(\mathbf{x}_n|\mathbf{x}_{n-1})$, which is used to propose those particles given the set of particles $\{\bar{\mathbf{x}}_{1:(n-1)}^{(p)}\}$ at the previous step $n-1$ that have been resampled from an SMC estimate of $p\left(\mathbf{x}_{1:(n-1)}|y_{1:(n-1)}\right)$. The weights corresponding to these particles are then calculated as follows:

$$W_n^{(p)} = \frac{p\left(\mathbf{x}_n^{(p)}\middle|\bar{\mathbf{x}}_{n-1}^{(p)}\right) p\left(y_n\middle|\mathbf{x}_n^{(p)}\right)}{q\left(\mathbf{x}_n^{(p)}\middle|\bar{\mathbf{x}}_{n-1}^{(p)}\right)}, \qquad w_n^{(p)} = \frac{W_n^{(p)}}{\sum_{i=1}^{P} W_n^{(i)}}, \qquad p = 1, \ldots, P.$$

Then, the proposal, resampling, and re-weighting steps are iterated for the next $n$.

In many applications of SMC inference, including probabilistic programming systems, the proposal distribution is taken to be the prior distribution $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ of the generative model. This simplifies the implementation of such systems as we can readily sample and evaluate the densities of the proposed values directly from the generative model.

One of the problems SMC methods suffer from is high variance of weights and the related problem of propagation of low weight particles that waste computation. This is because the propose-from-prior method usually gives proposals far from the optimal one since the prior is significantly different from the posterior. These problems can be mitigated by resampling, which intuitively resets the system at the expense of the increase in the immediate Monte Carlo variance. However, resampling can introduce another problem of degenerate particle trajectories, in which only a few distinct values are used to represent the early part of the inferred trajectories. Another approach to minimise the variance of the weights is to use proposal distributions $\hat{q}_n\left(\mathbf{x}_n|\mathbf{x}_{n-1}, y_{1:n}\right)$ that approximate the filtering distribution $p\left(\mathbf{x}_n|\mathbf{x}_{n-1}, y_{1:n}\right)$ [2], from which it is assumed impossible to sample directly because the normalisation factor requires a marginalisation over $\mathbf{x}_n$. In particular, there are no known ways to sample from these distributions directly in probabilistic programming systems. In this paper, we explore using information from previous runs to improve the proposal distribution so that it approximates the distribution $p\left(\mathbf{x}_n|\mathbf{x}_{n-1}, y_{1:n}\right)$.

## 2 Data-driven proposals for Sequential Monte Carlo

We want to have data-driven proposals for a certain subset of random choices $\mathcal{S} \subseteq \{1, \ldots, N\}$. We assume that the model prior $p\left(\mathbf{x}_n|\mathbf{x}_{n-1}\right)$ has the same structure for all $n \in \mathcal{S}$ so that we can learn the same proposals for all of them. Although it is impossible to sample from the optimal proposal—the filtering distribution $p\left(\mathbf{x}_n|\rho_n\right)$ where the environment $\rho_n := \left(\mathbf{x}_{n-1}, y_{1:n}\right)$—we can approximate it with a distribution $\hat{q}\left(\mathbf{x}_n|\mathfrak{N}_\theta(\phi(\rho_n))\right)$. This proposal distribution is parametrised by the output of some discriminative model $\mathfrak{N}_\theta$, which is in turn parametrised by $\theta$. The input of this model, namely features $\phi(\rho_n)$, is extracted from the environment $\rho_n$ by a fixed feature extractor function $\phi$.

To train the discriminative model, we need $M$ training inputs $\{\phi(\rho_n^{(i)})\}_{i=1}^M$ and related outputs $\{(\mathbf{x}_n^{(i)}, w_n^{(i)})\}_{i=1}^M$ such that each $\mathbf{x}_n^{(i)}$ is approximately drawn from the desired distribution $p\left(\mathbf{x}_n \middle| \rho_n^{(i)}\right)$. These weighted training inputs and outputs might be received from extensive sequential Monte Carlo inference on the training dataset.

The loss function used to obtain the parameters $\theta$ of the discriminative model is based on the Kullback-Leibler (KL) divergence. Taking the expectation of the KL-divergence with respect to $\mathbf{x}_{n-1}|y_{1:n}$, we can simplify the loss function $L_n(\theta)$ as follows:

$$\begin{aligned} L_n(\theta) &= \mathbb{E}_{p(\mathbf{x}_{n-1}|y_{1:n})}\left[D_{KL}\left(p\left(\mathbf{x}_n|\rho_n\right) \| \hat{q}\left(\mathbf{x}_n|\mathfrak{N}_\theta\left(\phi(\rho_n)\right)\right)\right)\right] \\ &= \mathbb{E}_{p(\mathbf{x}_{n-1}|y_{1:n})}\left[\mathbb{E}_{p(\mathbf{x}_n|\mathbf{x}_{n-1}, y_{1:n})}\left[\log \frac{p\left(\mathbf{x}_n|\rho_n\right)}{\hat{q}\left(\mathbf{x}_n|\mathfrak{N}_\theta(\phi(\rho_n))\right)}\right]\right] \\ &= \mathbb{E}_{p(\mathbf{x}_n|y_{1:n})}\left[\log \frac{p\left(\mathbf{x}_n|\rho_n\right)}{\hat{q}\left(\mathbf{x}_n|\mathfrak{N}_\theta(\phi(\rho_n))\right)}\right] = -\,\mathbb{E}_{p(\mathbf{x}_n|y_{1:n})}\left[\log \hat{q}\left(\mathbf{x}_n|\mathfrak{N}_\theta(\phi(\rho_n))\right)\right] + c. \end{aligned}$$

By ignoring the constant $c$ and substituting the Monte Carlo approximation of $p\left(\mathbf{x}_n|y_{1:n}\right)$, $\sum_{i=1}^M w_n^{(i)} \delta_{\mathbf{x}_n^{(i)}}(\mathbf{x}_n)$, we get that

$$L_n(\theta) \approx -\sum_{i=1}^M w_n^{(i)} \log \hat{q}\left(\mathbf{x}_n^{(i)} \middle| \mathfrak{N}_\theta(\phi(\rho_n))\right). \tag{1}$$

The loss function has a convenient form for neural networks because it can be decomposed to a sum of losses corresponding to each neural network output. We also note that in practice, we might use training outputs $\{\mathbf{x}_n^{(i)}\}_{i=1}^M$ not from the filtering distribution $p\left(\mathbf{x}_n \middle| \mathbf{x}_{n-1}^{(i)}, y_{1:n}^{(i)}\right)$, but from the smoothing distribution $p\left(\mathbf{x}_n \middle| \mathbf{x}_{n-1}^{(i)}, y_{1:N}^{(i)}\right)$. This is because most of statistical inference in existing probabilistic programming systems is directed towards the approximation of the smoothing distributions.

## 3 Experiments

### 3.1 Dependent Dirichlet process mixture of objects

For further experiments, we have chosen the dependent Dirichlet process mixture of objects (DDPMO) model [7] in order to demonstrate applicability of our approach to more complicated models. The DDPMO is a recent Bayesian non-parametric model for detection-free tracking and object modelling. The DDPMO models the position and colour $\mathbf{x}_{t,n}$ of a foreground pixel $n$ at a video frame $t$ as an observed variable. This observed variable $\mathbf{x}_{t,n}$ depends on the latent variables of the model such as cluster assignments $c_{t,1:N_t}$ and object parameters $\theta_t^k$ for each cluster $k$. The DDPMO is a native Bayesian non-parametric model since the number of clusters and the related object parameters is unbounded and dependent on the observed data. The generative process of the DDPMO is described in the appendix.

The DDPMO model was implemented as an Anglican program. The full model source code has 120 lines, with comments. In addition, the generalised Pólya urn (GPU) procedure was written as an Anglican program. Its source code has 70 lines, with comments, and can be used for any other model in the future. In order to make inference tractable, we implement and employ exchangeable random procedures (XRP) for conjugate priors, which are the essential part of the generative process in the DDPMO. These XRPs, implemented in Anglican, can also be re-used.

### 3.1.1 Data-driven inference for DDPMO

In DDPMO, we focus on improving a particular proposal of cluster assignment for a new data point (foreground pixel).

The features $\phi(\rho_k)$ of the environment $\rho_k = (\mathbf{x}_{k-1}, y_{1:k})$, which are the inputs to the neural network, consist of the following:

- Distances to the three nearest clusters in the ascending order, $d_i \in \mathbb{R}, i = 1, \ldots, 3$.
- Colour histograms of a $7 \times 7$ patch surrounding these three clusters in the discretised HSV space, normalised to sum to one, $h_i \in \mathbb{R}^{10}, \sum_{j=1}^{10} h_{ij} = 1$.
- Colour histogram of a $7 \times 7$ patch surrounding the new data point (i.e. pixel) in the discretised HSV space, normalised to sum to one, $c \in \mathbb{R}^{10}, \sum_{i=1}^{10} c_i = 1$.

The outputs of the neural network should be the probabilities of choosing one of existing clusters, or a completely new one. Thanks to the features $\phi(\rho_k)$ that identify the three closest clusters, we use the neural network with only five outputs. These outputs are the probabilities $p_{1:3}$ of choosing the three nearest clusters, the probability $p_4$ for the remaining $K-3$ existing clusters (so that each one has probability $p_4/(K-3)$), and the probability $p_5$ for the entirely new cluster. We directly set the weights of the proposal distribution $\hat{q}\left(\mathbf{x}_n|\eta\right)$ to the softmax output of the neural network. If $K < 3$, the prior proposal is used. If $K = 3$, the probability $p_4$ is set to zero (all other probabilities are re-normalised).

The cost function which is used for training of the neural network is the negative log probability given in (1). Noting that the weights are identical, we can use the negative log of the softmax output, and hence we can use neural network packages out of the box.

### 3.1.2 Football

For our experiments, a soccer video dataset was chosen [1]. This choice was made because the video contains many fast-moving, differently coloured and occluding objects. In addition, there exists a human-annotated ground-truth for this dataset. We select two subsequences of frames to form a training dataset and a test dataset. Both datasets mostly consist of moments of intensive play with many players on the field. To measure the performance, we use and report commonly used performance metrics: the sequence frame detection accuracy (SFDA) for object detection and the average tracking accuracy (ATA) for tracking [5].

At first, we run several iterations of SMC inference in Anglican for this model given the input frames from the formed training dataset, with 5000 particles. This allows us to extract inputs and outputs for the neural network, as described in the previous section. Then we train a neural network[1] using these extracted data. Once the neural network is trained, we run inference on the test frame sequences. We measure inference performance with three different types of proposals: the DDPMO prior proposal (i.e. just following the generative model), the data-driven proposal with the probabilities $p_{1:5}$ from the neural network, and the hand-tuned data-driven proposal with fixed probabilities $p_{1:5}$ from empirical analysis. The hand-tuned fixed probabilities $p_{1:5}$ approximate the distribution over the output over

[1]We used a feedforward neural network: one hidden layer with 10 nodes, `tansig` transfer function from the input to the hidden layer, softmax transfer function to the output, and cross-entropy error.

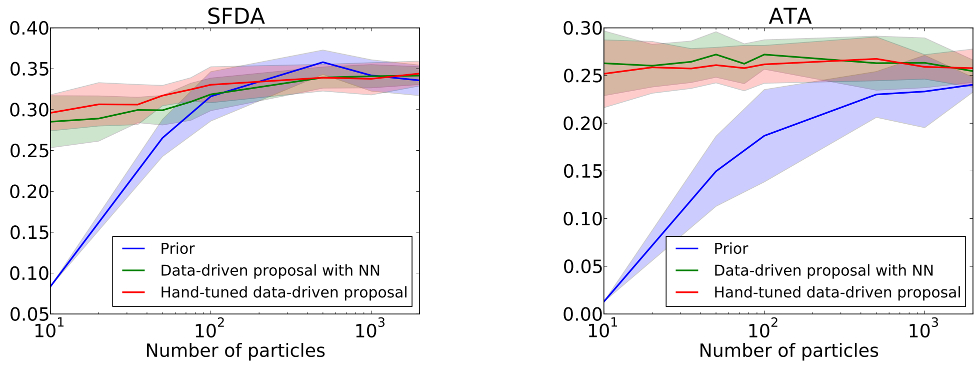

Figure 1: Sequence frame detection accuracy for object detection (SFDA) and the average tracking accuracy (ATA) metrics for different types of proposal distributions on the testing dataset. Y-axes refer to the values of SFDA and ATA metrics.

the three closest clusters, remaining old clusters, and a new cluster. Finally, we examine how the performance of these methods differ as we change the number of SMC particles.

Figure 1 illustrates experimental results. We get significant improvement in performance with the data-driven proposal. With 10 particles with the data-driven proposal, we get results similar to the results from running thousands of particles using the prior proposal. The data-driven proposal with the neural network show the same performance as the data-driven proposal with a hand-tuned discriminative model that always returns fixed $p_{1:5}$. Examples of frames with detected and tracked objects are provided in Figure 2 in the appendix.

## 4 Conclusion and Future Work

This abstract presents an approach to use data-driven proposals for Bayesian non-parametric models in probabilistic programming settings. Our experimental results show that the data-driven proposal significantly improves the inference perfomance. We assume that our proposal might be applied to non-parametric generative models that contain some distance function between clusters and data points (i.e. observations).

The data-driven proposal, which we presented, relies on the feature extractor. The feature extractor maps the current state of the unbounded number of clusters with their sufficient statistics to the input of the neural network. The feature extractor that we implement and use is also the significant part of the data-driven proposal. This is proved by the fact that the neural network performs as well as the fixed hand-tuned discriminative model. This is probably because the spatial factor is important for the model and the dataset, with which we worked. Therefore, there is future work to verify whether for more complex datasets and models data-driven proposals with neural networks provide more benefits.

Our work relates to other work in the field on data-driven proposals. The work on using discriminative proposals for Markov Chain Monte Carlo in parametric generative models include [8] and [4], with applications in computer vision. Recent work with sequential Monte Carlo includes neural adaptive SMC [3], where authors also adapt proposals by descending the inclusive Kullback-Leibler divergence between the proposal and the true posterior distributions on hidden variables given observations. They use recurrent neural networks to train proposal distributions for inference in parametric generative models with fixed dimensionality. Another related recent work is a new probabilistic programming language Picture [6], for which authors propose and describe how to use data-driven proposals for models in vision. They also use neural networks to learn proposals. To get the data to train the neural network, they sample both hidden variables and observations from the generative model unconditionally offline.

In future work, more recent versions of neural networks architectures can be applied to improve results by extracting better features and processing them more efficiently. In particular, one can think of using convolutional neural networks that process the part of the frame centered at the new observing pixel, or even the whole image.

**Acknowledgments**

We are very grateful to Willie Neiswanger, Jan Willem van de Meent and Brooks Paige for their help and advice.

## References

[1] T. D'Orazio, M. Leo, N. Mosca, P. Spagnolo, and P. L. Mazzeo. A semi-automatic system for ground truth generation of soccer video sequences. In *Advanced Video and Signal Based Surveillance, 2009. AVSS'09. Sixth IEEE International Conference on*, pages 559–564. IEEE, 2009.

[2] A. Doucet and A. M. Johansen. A tutorial on particle filtering and smoothing: Fifteen years later.

[3] S. Gu, R. E. Turner, and Z. Ghahramani. Neural adaptive sequential monte carlo. *arXiv preprint arXiv:1506.03338*, 2015.

[4] V. Jampani, S. Nowozin, M. Loper, and P. V. Gehler. The informed sampler: A discriminative approach to bayesian inference in generative computer vision models. *Computer Vision and Image Understanding*, 136:32–44, 2015.

[5] R. Kasturi, D. Goldgof, P. Soundararajan, V. Manohar, J. Garofolo, R. Bowers, M. Boonstra, V. Korzhova, and J. Zhang. Framework for performance evaluation of face, text, and vehicle detection and tracking in video: Data, metrics, and protocol. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(2):319–336, 2009.

[6] T. D. Kulkarni, P. Kohli, J. B. Tenenbaum, and V. K. Mansinghka. Picture: a probabilistic programming language for scene perception. 2015.

[7] W. Neiswanger, F. Wood, and E. P. Xing. The dependent dirichlet process mixture of objects for detection-free tracking and object modeling. 2014.

[8] Z. Tu and S.-C. Zhu. Image segmentation by data-driven markov chain monte carlo. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5):657–673, 2002.

[9] F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 2–46, 2014.

## Appendix

### 4.1 Object recognition and tracking results with the DDPMO

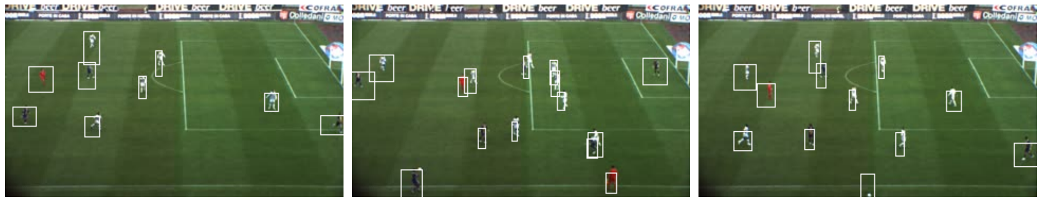

Figure 2: Frames of a soccer video dataset with detected and tracked objects using the DDPMO model in the probabilistic programming system Anglican.

### 4.2 The dependent Dirichlet process mixture of objects (DDPMO) model

The DDPMO models the position and colour $\mathbf{x}_{t,n}$ of a foreground pixel $n$ at a video frame $t$ as an observed variable. This observed variable $\mathbf{x}_{t,n}$ depends on the latent variables of the model such as cluster assignments $c_{t,1:N_t}$ and object parameters $\theta_t^k$ for each cluster $k$. Let $N_t$ be the total number of foreground pixels at a video frame $t$. Then the generative process for each time step $t = 1, \ldots, T$ is as follows:

1. Draw $\{c_{t,1:N_t}, K_{t,N_t}, m_{t,0}^{1:K_{t-1,N_{t-1}}}\} \sim \mathrm{GPU}(\alpha, \rho)$.
2. For $k = 1, \ldots, K_{t,N_t}$:
$$\text{draw } \theta_t^k \sim \begin{cases} \mathrm{T}(\theta_{t-1}^k) & \text{if } k \leq K_{t-1,N_{t-1}} \\ \mathbb{G}_0(\boldsymbol{\eta}_0) & \text{if } k > K_{t-1,N_{t-1}}. \end{cases}$$
3. For $n = 1, \ldots, N_t$: draw $\mathbf{x}_{t,n} \sim \mathrm{F}(\theta_t^{c_{t,n}})$.

Here, $K_{t,n}$ represents the total number of clusters after processing pixel $\mathbf{x}_{t,n}$ and $m_{t,n}^k$ represents the size of cluster $k$ at time after processing $\mathbf{x}_{t,n}$. $\mathrm{F}(\theta_t^k)$ is a generative model that generates the foreground pixels given the object parameter $\theta_t^k$ of cluster $k$. Distributions T and $\mathbb{G}_0$ are transition and prior distributions of the object parameters $\theta_t^k$, which must satisfy a technical condition of the GPU, such that $\int \mathbb{G}_0(\theta_{t-1}^k)\, \mathrm{T}\left(\theta_t^k \middle| \theta_{t-1}^k\right) \mathrm{d}\theta_{t-1}^k = \mathbb{G}_0(\theta_t^k)$. The dependent Dirichlet process prior $\mathrm{GPU}(\alpha, \rho)$ is parametrised by the birth and deletion rates $\alpha$ and $\rho$, which, for each time $t$, governs the evolution of the number of clusters. In this model, the target quantity to infer is
$$p\left(\{c_{t,1:N_t}, K_{t,N_t}, \theta_t^{1:K_{t,N_t}}\}_{t=1:T} \middle| \{\mathbf{x}_{t,n}\}_{t=1:T,n=1:N_t}\right).$$

### 4.3 Generalised Pólya urn

The generative model of the GPU at each time step $t$ is as follows:

1. For $k = 1, \ldots, K_{t-1,N_{t-1}}$:
   (a) Draw $\Delta m_{t-1}^k \sim \mathrm{Binomial}(m_{t-1,N_{t-1}}^k, \rho)$.
   (b) Set $m_{t,0}^k = m_{t-1,N_{t-1}}^k - \Delta m_{t-1}^K$.
2. For $n = 1, \ldots, N_t$
   (a) Draw $c_{t,n} \sim \mathrm{Categorical}\left(\frac{m_{t,n-1}^1}{\alpha + \sum_k m_{t,n-1}^k}, \ldots, \frac{m_{t,n-1}^{K_{t,n-1}}}{\alpha + \sum_k m_{t,n-1}^k}, \frac{\alpha}{\alpha + \sum_k m_{t,n-1}^k}\right)$.
   (b) If $c_{t,n} \leq K_{t,n-1}$, set $m_{t,n}^{c_{t,n}} = m_{t,n-1}^{c_{t,n}} + 1$, $m_{t,n}^{\setminus c_{t,n}} = m_{t,n-1}^{\setminus c_{t,n}}$ and $K_{t,n} = K_{t,n-1}$;[2]
   (c) Otherwise, set $m_{t,n}^{c_{t,n}} = 1, m_{t,n}^{\setminus c_{t,n}} = m_{t,n-1}^{\setminus c_{t,n}}$ and $K_{t,n} = K_{t,n-1} + 1$.

[2] $m_{t,n}^{\setminus c_{t,n}}$ is the set $\{m_{t,n}^1, \ldots, m_{t,n}^{K_{t,n}}\} \setminus \{m_{t,n}^{c_{t,n}}\}$.

## 4.4 The DDPMO code in Anglican

```
(ns ddpmo.ddpmo
  (:use [anglican emit runtime]
        [anglib xrp utils new-dists anglican-utils]
        ddpmo.ddpmo-header)
  (:require [clojure.core.matrix :as m]
            [clojure.core.matrix
             :refer [identity-matrix mmul add sub transpose matrix to-nested-vectors]
             :rename {identity-matrix eye
                      add madd
                      sub msub
                      transpose mtranspose}]
            [clojure.core.matrix.linear :as ml]))

(with-primitive-procedures
  [multivariate-t mvn-conjugate-fast dirichlet-multinomial-process
   DIRICHLET-MULTINOMIAL-PROCESS-STATE-INFO MVN-PROCESS-FAST-STATE-INFO
   matrix produce-matrix-from-vector to-nested-vectors mtranspose matrix-to-clojure-vector]

  (defquery ddpmo
    "The Dependent Dirichlet Process Mixture of Objects for Detection-free Tracking"
    [data Nts proposal-type]

    (let [

          ;;;;;; DDPMO model ;;;;;;

          ;; Hyperparameters for squares/objects/football
          alpha 0.1 ; for GPU
          rho 0.32 ; for GPU
          mu-0 (produce-matrix-from-vector [0 0]) ; for normal-inverse-wishart
          k-0 0.05 ; for normal-inverse-wishart
          nu-0 10 ; for normal-inverse-wishart (old prior 60)
          Lambda-0 (matrix [[0.7 0] [0 0.07]])
          q-0 (vec (repeat 10 10.0)) ; for Dirichlet.
                                     ; The dimensionality must match number of RGB bins V
          M 10 ; for GO (eqns (7-8))
          multinomial-trials 49 ; for eqn (2)... this is m x m where m = 2L + 1

          extract-old-style-theta
          (fn [theta]
            (let
              [mvn-process (retrieve (get theta 'positions))
               dirichlet-multinomial-process-instance (retrieve (get theta 'colours))
               mu-Sigma (MVN-PROCESS-FAST-STATE-INFO mvn-process)
               ps (DIRICHLET-MULTINOMIAL-PROCESS-STATE-INFO
                   dirichlet-multinomial-process-instance)
               theta]
              {'mu (get mu-Sigma 'mu) 'Sigma (get mu-Sigma 'Sigma)
               'trials multinomial-trials 'ps ps}))

          get-N (fn [t] (nth Nts (dec t)))

          ;; Transition distribution
          T (fn T [prev-theta]
              (let [previous-mvn-process
                      (get prev-theta 'positions)
                    previous-dirichlet-multinomial-process (get prev-theta 'colours)

                    new-mvn-process (XRP (mvn-conjugate-fast mu-0 k-0 nu-0 Lambda-0))
                    new-dirichlet-multinomial-process
                      (XRP (dirichlet-multinomial-process q-0 multinomial-trials))

                    ;; Auxiliary transition
```

```
                  _ (repeatedly M (fn []
                                    (INCORPORATE new-mvn-process
                                                 (SAMPLE previous-mvn-process))))
                  _ (repeatedly M (fn []
                                    (INCORPORATE
                                       new-dirichlet-multinomial-process
                                       (SAMPLE previous-dirichlet-multinomial-process))))
                  ]
              {'positions new-mvn-process 'colours new-dirichlet-multinomial-process}))

    ;; Base distribution
    G0 (fn G0 []
         (let [mvn-process
                 (XRP (mvn-conjugate-fast mu-0 k-0 nu-0 Lambda-0))
               dirichlet-multinomial-process-instance
                 (XRP (dirichlet-multinomial-process q-0 multinomial-trials))
               ]
          {'positions mvn-process 'colours dirichlet-multinomial-process-instance}))

    [gpu get-theta] (create-gpu alpha rho G0 T get-N)

    ;; Helper function
    ;; Returns parameters for the corresponding table of foreground pixel n at time t
    get-theta-t-n (mem (fn get-theta-t-n [t n]
                         (let [customers (gpu t n)
                               cs (get customers 'cs)
                               k (get cs (dec n))]
                           (get-theta t k))))

    ;;;;;; OBSERVES ;;;;;;
    observe-lines
    (fn observe-lines [lines line-id]
      (if (nil? (first lines))
        true
        (let [line (first lines)
              pos (get line 'pos)
              _ (store "current-pos" (matrix-to-clojure-vector pos))
              col (get line 'col)
              _ (store "current-col" col)
              t (get line 't)
              n (get line 'n)
              theta (get-theta-t-n t n)
              positions-process (get theta 'positions)
              colours-process (get theta 'colours)]

          ; Observing positions.
          (OBSERVE positions-process pos)

          ; Observing colours
          (OBSERVE colours-process col)

          (if (= n (get-N t))
            (let [gpu (gpu t n)
                  cs (get gpu 'cs)
                  relevant-clusters (distinct cs)
                  thetas (map (fn [k]
                                (let [theta (get-theta t k)
                                      theta (extract-old-style-theta theta)
                                      mu (get theta 'mu)
                                      Sigma (get theta 'Sigma)
                                      ps (get theta 'ps)]
                                  {'k k 'mu mu 'Sigma Sigma 'ps ps}))
                              relevant-clusters)
                  res {'t t 'n n 'gpu gpu 'thetas thetas}]
              (predict res)))
```

```
                (observe-lines (rest lines) (inc line-id)))))]

      (observe-lines data 0))))

(defn -main [data-set-name number-of-particles num-particles-to-output
             proposal-type & ignore-following-args]
  (let [number-of-particles (parse-int number-of-particles)
        num-particles-to-output (parse-int num-particles-to-output)
        proposal-type (str proposal-type)
        _ (case proposal-type "prior" :okay "handtuned" :okay "nn" :okay)
        [data Nts] (load-DDPMO-data data-set-name)
        query-results (doquery :smc ddpmo [data Nts proposal-type]
                               :number-of-particles number-of-particles)
        results
        (doall
         (map
          (fn [particle-output particle-id]
            (doall
             (map
              (fn [x]
                (println (str particle-id "," (first x) "," (second x) ",0.0")))
              (get particle-output :anglican.state/predicts))))
          (take num-particles-to-output query-results)
          (range num-particles-to-output)))]
    results))
```

### 4.5 The GPU code in Anglican

```
;;;;;; GPU definition ;;;;;;

; Creates an instance of a GPU process.
; Takes:
; * GPU's alpha and rho.
; * Base distribution G0.
; * Transition distribution T.
; * function get-N which returns the number of points at each time.
; Returns: [gpu get-theta]
(defm create-gpu [alpha rho G0 T get-N]
  (let
    [;; Given vector of table sizes ms = [m1 m2 ...], returns a new vector of table
     ;; sizes by removing customers from tables with probability rho
     remove-customers
     (fn [ms]
       (vec (map (fn [m]
                   (if (= m 0) 0 (- m (SAMPLE (binomial m rho)))))
                 ms)))

     ;; Returns {'cs (vector of n cluster ids) 'K (number of unique clusters at n)
     ;;          'ms (vector of cluster sizes at n)}
     ;; after processing foreground pixel n at time t
     ;; n goes from 1
     ;; c_i goes from 0
     ;; K = max(c_i) + 1
     ;; t goes from 1
     gpu (mem
          (fn gpu [t n]
            (if (= n 0)
              ;; Initialise
              (if (= t 1)
                {'cs '[] 'K 0 'ms '[]}
                (let [prev-t-gpu (gpu (dec t) (get-N (dec t)))
                      prev-K (get prev-t-gpu 'K)
                      prev-ms (get prev-t-gpu 'ms)]
                  {'cs '[] 'K prev-K 'ms (remove-customers prev-ms)}))

              ;; Get from step (n - 1)
              (let [prev-n-gpu (gpu t (dec n))
                    cs (get prev-n-gpu 'cs)
                    K (get prev-n-gpu 'K)
                    ms (get prev-n-gpu 'ms)
                    w (conj ms alpha)
                    c (SAMPLE (discrete w))
                    new-cs (conj cs c)
                    new-K (max K (inc c))
                    new-ms (assoc ms c (inc (get ms c 0)))]
                {'cs new-cs 'K new-K 'ms new-ms}))))

     ;; Returns parameters for table k at time t using either
     ;; transition distribution T or base distribution G0
     get-theta (mem (fn get-theta [t k]
                      (if (= t 1)
                        (G0)
                        (let [prev-customers (gpu (dec t) (get-N (dec t)))
                              prev-K (get prev-customers 'K)
                              initial-ms (get (gpu t 0) 'ms)]
                          (if (> k (dec prev-K))
                            (G0)
                            (if (= (nth initial-ms k) 0)
                              nil
                              (T (get-theta (dec t) k))))))))]
    [gpu get-theta]))
```

## 4.6 Clojure code for the data-driven proposal

```
(def NUMBER-OF-NEAREST-CLUSTERS 3)

(def sort-thetas
  (fn [thetas]
    (let
      [my-comparer
       (fn [el1 el2]
         (< (nth el1 2) (nth el2 2)))]
      (sort my-comparer thetas))))

(def distance
  (fn [[x1 y1] [x2 y2]]
    "Returns Euclidean distance between two 2D points."
    ; Important! Here x is really y, and vice versa.
    ; This is because in the MATLAB code the first coordinate is y.
    (pow (+ (pow (- x1 x2) 2.0) (pow (- y1 y2) 2.0)) 0.5)))
```

## 4.7 Anglican code (within the DDPMO model) for the data-driven proposal

```
get-thetas
(fn [t n]
  "Returns thetas for active clusters (ms[i] > 0)
  at data point (t, n). This function should be
  called only when we already processed that data point."
  (let
    [
     gpu-state (gpu t n)
     ms (get gpu-state 'ms)
     get-theta (fn [t k] (if (> (nth ms k) 0) (get-theta t k) nil))
     thetas (map (fn [k] (list k (get-theta t k))) (range (count ms)))
     thetas (filter (fn [el] (not (nil? (second el)))) thetas)
     ]
    thetas))

```

```
get-mean-coords
(fn [theta]
  "Extracts mean from the theta as Clojure vector."
  (let
    [coords (matrix-to-clojure-vector
               (get (MVN-PROCESS-FAST-STATE-INFO
                       (retrieve (get theta 'positions))) 'mu))]
    coords))

get-nearest-thetas
(fn [t n [x y]]
  "Gets an ordered list of theta which are the nearest to the point [x y]
  based on the state at the previous data point (t, n - 1)."
  (if (and (= t 1) (= n 1))
    nil
    (let
      [[t n]
       (if (= n 1)
         [(- t 1) (get-N (- t 1))]
         [t (- n 1)])]
      (let
        [thetas (get-thetas t n)
         thetas (map (fn [[k theta]]
                       (list k theta (distance [x y]
                                               (get-mean-coords theta)))) thetas)
         thetas (sort-thetas thetas)
         thetas (take NUMBER-OF-NEAREST-CLUSTERS thetas)]
        (if (< (count thetas) NUMBER-OF-NEAREST-CLUSTERS)
          nil
          thetas)))))

;; Do the trick to allow mutual recursion.
_ (store "get-nearest-thetas" get-nearest-thetas)
```

### 4.8 Code for the GPU, to get data for the proposal for train datasets

```
NEAREST-THETAS ((retrieve "get-nearest-thetas") t n (retrieve "current-pos"))
for-proposal
  (map
   (fn [the-list]
     (let
       [theta-id (nth the-list 0)
        theta (nth the-list 1)
        distance-to-the-center (nth the-list 2)]
       (list
        theta-id
        (DIRICHLET-MULTINOMIAL-PROCESS-STATE-INFO (retrieve (get theta 'colours)))
        distance-to-the-center)))
   NEAREST-THETAS)
nn-input
  (concat
   (apply concat (doall (map (fn [data]
                             (concat (nth data 1) (list (nth data 2)))) for-proposal)))
   (doall (map (fn [x] (/ x 49.0)) (retrieve "current-col")))))
c (if (or (not (= (count nn-input) 43)) (= proposal-type "prior"))
    (SAMPLE (discrete w))
    (let
      [dist (sample-cluster-id nn-input w 0.8
                             (map first NEAREST-THETAS) (= proposal-type "handtuned"))
       [my-sample log-likelihood] (sample dist)]
      (add-log-weight log-likelihood)
      my-sample))
```

### 4.9 Code for the GPU, to use the proposal for test datasets

```
NEAREST-THETAS ((retrieve "get-nearest-thetas") t n (retrieve "current-pos"))
for-proposal
  (map
   (fn [the-list]
     (let
       [theta-id (nth the-list 0)
        theta (nth the-list 1)
        distance-to-the-center (nth the-list 2)]
       (list
        theta-id
        (DIRICHLET-MULTINOMIAL-PROCESS-STATE-INFO (retrieve (get theta 'colours)))
        distance-to-the-center)))
   NEAREST-THETAS)
_ (predict (list for-proposal c (retrieve "current-col") (count w)))
```