2nd Symposium on Advances in Approximate Bayesian Inference, 2019 1–16

# Divide, Conquer, and Combine: a New Inference Strategy for Probabilistic Programs with Stochastic Support

**Yuan Zhou** YUAN.ZHOU@CS.OX.AC.UK
*Department of Computer Science, University of Oxford*

**Hongseok Yang** HONGSEOK.YANG@KAIST.AC.KR
*School of Computing, KAIST*

**Yee Whye Teh** Y.W.TEH@STATS.OX.AC.UK
*Department of Statistics, University of Oxford*

**Tom Rainforth** RAINFORTH@STATS.OX.AC.UK
*Department of Statistics & Christ Church, University of Oxford*

## Abstract

Universal probabilistic programming systems (PPSs) provide a powerful framework for specifying rich and complex probabilistic models. However, this expressiveness comes at the cost of substantially complicating the process of drawing inferences from the model. In particular, inference can become challenging when the support of the model varies between executions. Though general-purpose inference engines have been designed to operate in such settings, they are typically inefficient, often relying on proposing from the prior to make transitions. To address this, we introduce a new inference framework: **Divide, Conquer, and Combine (DCC)**. DCC divides the program into separate straight-line sub-programs, each of which has a fixed support allowing more powerful inference algorithms to be run locally, before recombining their outputs in a principled fashion. We show how DCC can be implemented as an automated and general-purpose PPS inference engine, and empirically confirm that it can provide substantial performance improvements over previous approaches.

## 1. Introduction

*Universal PPSs*, such as Church (Goodman et al., 2008), Venture (Mansinghka et al., 2014), Anglican (Wood et al., 2014) and Pyro (Bingham et al., 2018), are set up to try and support the widest possible range of models a user might wish to write. Though this means that such systems can be used to write models which would be otherwise difficult to encode, this expressiveness comes at the cost of significantly complicating the automation of inference. In particular, models may contain variables with mixed types or have varying, or even unbounded, dimensionalities; characteristics which cause significant challenges at the inference stage. In this paper, we aim to address one of the most challenging of these complicating factors: variables whose very existence is stochastic, often, though not always, leading to the overall dimensionality of the model varying between realizations.

Some very basic inference algorithms, such as importance sampling from the prior, are able to deal with this problem naturally, but they are catastrophically inefficient for all but the most simple models. Sequential Monte Carlo (Wood et al., 2014) and variational (Paige, 2016) approaches can sometimes also be applied, but only offer improvements for models with particular exploitable structures. MCMC approaches, on the other hand, are difficult

to apply due to the need to construct proposals able to switch between the different variable configurations, something which is difficult to achieve even in a problem specific manner, let alone automate for generic problems. Moreover, ensuring these proposals remain efficient can be almost impossible, as different configurations might not have natural similarities or "neighboring regions"; the problem is analogous to running MCMC on a highly multi-modal distribution without any knowledge of where the different modes are. In short, there are a wide range of models for which no effective PPS-suitable inference methods currently exist. More discussion can be seen in Appendix B.

To this end, we introduce a new framework—***Divide, Conquer, and Combine*** (DCC)—for performing inference in such models. DCC works by ***dividing*** the program into separate straight-line sub-programs with fixed support, ***conquering*** these separate sub-problems using an inference strategy that exploits the fixed support to remain efficient, and then ***combining*** the resulting sub-estimators to an overall approximation of the posterior. By splitting the original program up into its separate configurations, we effectively transfer this transitioning problem to one of estimating the marginal likelihood for the different models, something which is typically much easier to achieve. Furthermore, this approach also allows us to introduce meta-strategies for allocating resources between sub-problems, thereby explicitly controlling the exploration-exploitation trade-off in a manner akin to Rainforth et al. (2018); Lu et al. (2018). To demonstrate its potential utility, we implement a specific realization of our DCC framework as an automated and general-purpose inference engine in the PPS Anglican (Wood et al., 2014), finding that it is able to achieve substantial performance improvements and tackle more challenging models than existing approaches.

## 2. Divide, Conquer, and Combine

To aid exposition and formalize these programs, we will focus on the particular universal PPS Anglican (Wood et al., 2014), but note that our ideas are applicable to any universal PPS for which the program's support is not necessarily fixed. The density of an Anglican program is derived by executing it in a forward manner, drawing from `sample` statements when encountered, and keeping track of density components originating from both the `sample` and `observe` terms. Specifically, let $\{x_i\}_{i=1}^{n_x} = x_1, \ldots, x_{n_x}$ represent the random variables generated from the encountered `sample` statements, where the $i$-th encountered `sample` statement has a lexical program address $a_i$. More formal definition of the density is provided in Appendix A.1. For clarity, we refer to the sequence $a_{1:n_x}$ as the ***path*** of a trace and $x_{1:n_x}$ as the ***draws***. A program with ***stochastic support*** means that the path $a_{1:n_x}$ of the program varies between different realizations: a different value for the path corresponds to a different *configuration* of variables being sampled.

Unlike most existing inference approaches which directly target the full program density, DCC breaks the problem into individual sub-problems with fixed support and tackles them separately. Specifically, it ***divides*** the overall program into separate straight-line sub-programs according to their execution paths, ***conquers*** each sub-program by running inference locally, and ***combines*** the results together in a principled manner.

**Divide** The first step of DCC is to divide the given probabilistic program into its constituent straight-line programs (SLPs), where each SLP is a partition of the overall program

```
1  (let [z0 (sample (normal 0 2))
2        y1 9]
3    (if (< z0 0)
4        (let [z1 (sample (normal -5 2))]
5           (observe (normal z1 2) y1)
6           [z0 z1])
7        (let [z2 (sample (normal 5 2))
8              z3 (sample (normal z2 2))]
9           (observe (normal z3 2) y1))
10          [z0 z2 z3])))
```

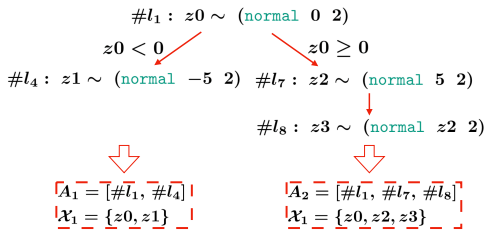



Figure 1: A branching model written in Anglican (left) and its execution trace (right).

corresponding to a particular sequence of sample addresses encountered during execution, i.e. a particular path $a^k_{1:n_{x,k}}$. Each SLP has a fixed support as the set of variables $x^k_{1:n_{x,k}}$ it draws are fixed by the path, i.e. the program always draws from the same set of sample statements in the same fixed order. Using the shorthand $A_k := a^k_{1:n_{x,k}}$, the set of of all possible execution paths is now given by $\{A_k\}_{k=1}^K$, where $K$ must be countable (but may not be finite) and $k$ indexes the individual SLPs (the ordering of which is inconsequential). For the example in Figure 1, this set consists of two paths $A_1 = [\#l_1, \#l_4]$ and $A_2 = [\#l_1, \#l_7, \#l_8]$, where we use $\#l_j$ to denote the lexical address of the `sample` statement at the $j^{\text{th}}$ line.

Dividing a program into its constituent SLPs implicitly partitions the overall target density into disjoint regions, with each part defining a sub-model on the corresponding sub-space. The density $\pi_k(x)$ of the SLP $A_k$ is defined with respect to the variables $\{x_i\}_{i=1}^{n_{x,k}}$ that are paired with the addresses $\{a_i\}_{i=1}^{n_{x,k}}$ of $A_k$, which we only have access to the unnormalized version $\gamma_k(x)$. We use $\mathcal{X}_k$ to denote the corresponding sub-space of $x_{1:n_{x,k}}$ and note that the union of $\mathcal{X}_k$ for all $k$ is the entire latent space defined by the overall program, $\mathcal{X} = \bigcup_{k=1}^K \mathcal{X}_k$. Unlike previously, $n_{x,k}$ and $A_k$ are now, critically, *deterministic* variables so that the support of the sub-model is *fixed*. The formal definition of the density for a SLP is given in Appendix A.2. Following our example in Figure 1, SLP $A_1$ corresponds to the sub-space $\mathcal{X}_1 = \{[x_1, x_2] \in \mathbb{R}^2 \mid x_1 < 0\}$ and has the density $\gamma_1(x) = \mathcal{N}(x_1; 0, 2)\mathcal{N}(x_2; -5, 2)\mathcal{N}(y_1; x_2, 2)\mathbb{I}[x_1 < 0]$, while $A_2$ corresponds to the sub-space $\mathcal{X}_2 = \{[x_1, x_2, x_3] \in \mathbb{R}^3 \mid x_1 \geq 0\}$ and has density $\gamma_2(x) = \mathcal{N}(x_1; 0, 2)\mathcal{N}(x_2; 5, 2)\mathcal{N}(x_3; x_2, 2)\mathcal{N}(y_1; x_3, 2)\mathbb{I}[x_1 \geq 0]$. More details are in Appendix C.2.

**Conquer** Given the set of SLPs produced by the divide step, we now carry out the required local inference for each. This forms our conquer step and its aim is to produce a set of estimates for the individual SLP densities $\pi_k(x)$ and corresponding marginal likelihoods $Z_k$. As each SLP has a fixed support, this can be achieved with conventional inference approaches, with a large variety of methods potentially suitable. Note that $\pi_k(x)$ and $Z_k$ need not be estimated using the same approach, e.g. we may use an MCMC scheme to estimate $\pi_k(x)$ and then introduce a separate estimator for $Z_k$. In short, we will propose the use of a combination of Metropolis-within-Gibbs (MwG) and the parallel interacting Markov adaptive importance sampling (PI-MAIS) algorithm of Martino et al. (2017) for performing the local inference with further details in Appendix C.1.

An important component in carrying out this conquer step effectively, is to note that it is not usually necessary to obtain equally high fidelity estimates for each SLP. Specifically,

SLPs with small marginal likelihoods $Z_k$ only make a small contribution to the overall density and thus do not require as accurate estimation as SLPs with large $Z_k$s. As such, it will typically be beneficial to carry out *resource allocation* as part of the conquer step, that is, to generate our estimates in an online manner where at each iteration we use information from previous samples to decide the best SLP(s) to update our estimates for. Further details on this, along with our suggested approach for the local inference itself, are given in and C.3.

**Combine** The last stage of DCC is to combine the local estimates from the individual SLPs to an overall estimate of the conditional distribution for the original program. For this, we can simply note that, because the supports of the individual SLPs are disjoint and their union is the complete program, we have $\gamma(x) = \sum_{k=1}^{K} \gamma_k(x)$ and $Z = \sum_{k=1}^{K} Z_k$, such that the unnormalized density and marginal likelihoods are both additive. We then have

$$\pi(x) = \frac{\sum_{k=1}^{K} \gamma_k(x)}{\sum_{k=1}^{K} Z_k} = \frac{\sum_{k=1}^{K} Z_k \pi_k(x)}{\sum_{k=1}^{K} Z_k} \approx \frac{\sum_{k=1}^{K} \hat{Z}_k \hat{\pi}_k(x)}{\sum_{k=1}^{K} \hat{Z}_k} \tag{1}$$

where $\hat{\pi}_k(x)$ and $\hat{Z}_k$ are the SLP estimates generated during the conquer step.

When using an MCMC sampler for $\pi_k(x)$, $\hat{\pi}_k(x)$ will take the form of an empirical measure comprising of a set of samples, i.e. $\hat{\pi}_k(x) = \frac{1}{N_k} \sum_{m=1}^{N_k} \delta_{\hat{x}_m^k}(x)$. If we instead use an importance sampling or particle filtering based approach, our empirical measure will instead compose of weighted samples. We note that in this case, the $\hat{Z}_k$ term in the numerator of (1) will cancel with any potential self-normalization term used in $\hat{\pi}_k(x)$, such that we can instead think of using the estimate $\pi(x) \approx (\sum_{k=1}^{K} \hat{\gamma}_k(x))/(\sum_{k=1}^{K} \hat{Z}_k)$.

Specific strategies for implementing each component are introduced in Appendix C.

## 3. Experiments

We compare DCC against Anglican's importance sampling (IS) and Random-walk Metropolis Hasting (RMH), a variant of the Lightweight Metropolis Hasting (LMH) (Wingate et al., 2011), on two models given the same computational resources over 15 runs.

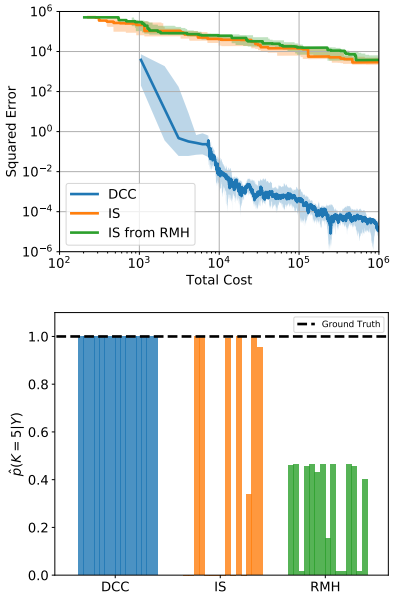
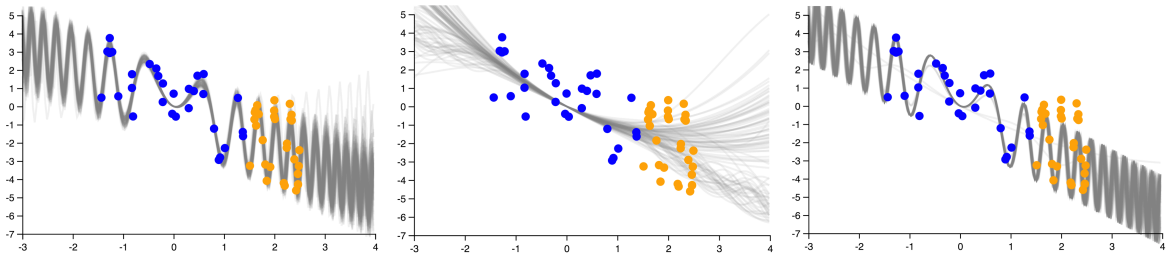



Figure 2: GMM

**Gaussian Mixture model (GMM)** The first model is a GMM where the number of the mixtures as well as the mean of each mixture are unknown. We first examine the convergence of the overall log marginal likelihood $Z$, and present the median (solid line) and $25\% - 75\%$ quantiles (shaded area) of the squared error of the estimates in Figure 2 (top) among three methods. As RMH cannot be used directly here, we instead draw importance samples centered around the RMH chain in a manner akin to PI-MAIS (Martino et al., 2017). It shows that DCC outperforms both baselines by many orders of magnitude. To further investigate, we look into the posterior distribution of $K$ and compare the estimates of $\hat{p}(K = 5 \,|\, y_{1:N_y})$ in Figure 2 (bottom). DCC performs



(*a*) DCC (ours) (*b*) IS (*c*) RMH

Figure 3: Estimates of the posterior distribution $p(\Theta|D)$ in the function induction model.

the most accurately and consistently whereas IS occasionally gives reasonable estimates and RMH has a tendency to get stuck in one sub-model. More details are given in Appendix D.1.

**Function Induction** The second model is about function induction generated by a Probabilistic Context Free Grammar (PCFG) (Manning et al., 1999). We specify the structure of a candidate function using a PCFG and distribution over the function parameters, and estimate the posterior of both for given data. Our PCFG model consists of four production rules:

$$R = \{e \to x \mid x^2 \mid \sin(a * e) \mid a * e + b * e\}$$

where $x$ and $x^2$ are terminal symbols, $a$ and $b$ are unknown coefficient parameters, and $e$ is a non-terminal symbol. The rules have fixed probabilities $p_R$. The model also has the prior distributions for each parameter.

Conditioned on some training data, we want to infer the posterior distribution of the function structure as well the underlying parameters, which can be used to do prediction given the test data. We report the mean and one standard derivation of the test log marginal likelihood (LML) estimates (the higher the better) in Table 1 and DCC outperforms the baselines both in terms of predictive accuracy and stability. A more qualitatively comparison of the posterior distribution are provided in Figure 3. DCC samples capture the periodicity of the training data and in general interpolates them well, while remaining uncertain in the regions of no data. Though RMH does find some good functions, it becomes stuck in a particular mode and doe not fully capture the uncertainty in the model, leading to poor predictive performance. See more results in Appendix D.2.

| | LML |
|---|---|
| DCC | **-14.483 ± 0.219** |
| IS | -213.642 ± 0.335 |
| RMH | -19.870 ± 7.262 |

Table 1: LML comparison.

## 4. Conclusions

In this paper, we have proposed *Divide, Conquer and Combine (DCC)*, a new inference strategy for probabilistic programs with stochastic support. We have shown that by breaking down the overall inference problem into a number of separate inferences of subprograms of fixed support, the DCC framework can provide substantial performance improvements over existing approaches which directly target the full program. To realize this potential, we have shown how to implement a particular instance of DCC as an automated engine in the PPS Anglican, and demonstrated its effectiveness through two example problems.

**Acknowledgements**

YZ is sponsored by China Scholarship Council (CSC). HY was supported by the Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and also by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068177). YWT's and TR's research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ ERC grant agreement no. 617071. TR is also supported in part by Junior Research Fellowship from Christ Church, University of Oxford and and in part by EPSRC funding under grant EP/P026753/1.

## References

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

Alexandra Carpentier, Remi Munos, and András Antos. Adaptive strategy for stratified monte carlo sampling. *Journal of Machine Learning Research*, 16:2231–2271, 2015.

Arun Chaganty, Aditya Nori, and Sriram Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics*, pages 153–160, 2013.

Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

David Duvenaud, James Robert Lloyd, Roger Grosse, Joshua B Tenenbaum, and Zoubin Ghahramani. Structure discovery in nonparametric regression through compositional kernel search. *arXiv preprint arXiv:1302.4922*, 2013.

Noah D Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages, 2014.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A Language for Generative Models. In *In UAI*, pages 220–229, 2008.

Peter J Green. Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

Peter J Green. Trans-dimensional markov chain monte carlo. *Oxford Statistical Science Series*, pages 179–198, 2003.

Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1945–1954. JMLR. org, 2017.

Tuan Anh Le. Inference for higher order probabilistic programs. *Masters thesis, University of Oxford*, 2015.

Xiaoyu Lu, Tom Rainforth, Yuan Zhou, Jan-Willem van de Meent, and Yee Whye Teh. On exploration, exploitation and learning in adaptive importance sampling. *arXiv preprint arXiv:1810.13296*, 2018.

Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

Luca Martino, Victor Elvira, David Luengo, and Jukka Corander. Layered adaptive importance sampling. *Statistics and Computing*, 27(3):599–623, 2017.

Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.

Aditya Nori, Chung-Kil Hur, Sriram Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

Timothy Brooks Paige. *Automatic inference for higher-order probabilistic programs*. PhD thesis, University of Oxford, 2016.

Tom Rainforth. *Automating Inference, Learning, and Design using Probabilistic Programming*. PhD thesis, University of Oxford, 2017.

Tom Rainforth, Yuan Zhou, Xiaoyu Lu, Yee Whye Teh, Frank Wood, Hongseok Yang, and Jan-Willem van de Meent. Inference trees: Adaptive inference with exploration. *arXiv preprint arXiv:1806.09550*, 2018.

Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. C3: lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, pages 28–37, 2016. URL http://proceedings.mlr.press/v51/ritchie16.html.

David Roberts, Marcus Gallagher, and Thomas Taimre. Reversible jump probabilistic programming. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 634–643, 2019.

David Tolpin, Jan-Willem van de Meent, Brooks Paige, and Frank D. Wood. Output-sensitive adaptive metropolis-hastings for probabilistic programs. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II*, pages 311–326, 2015. doi: 10.1007/978-3-319-23525-7_19. URL https://doi.org/10.1007/978-3-319-23525-7_19.

David Wingate, Andreas Stuhlmüller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

F. Wood, J. W. van de Meent, and V. Mansinghka. A New Approach to Probabilistic Programming Inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

## Appendix A. Formal Definition of the Density of a Probabilistic Program

### A.1. Density of the Probabilistic Program in Anglican

Anglican inherits its general syntax from Clojure, extending this with two special forms: `sample` and `observe`, between which the distribution of the program is defined. `sample` statements are used to draw random variables from provided probability distributions, while `observe` statements are used to condition on data. Informally, they can be respectively thought of as prior and likelihood terms.

The density of an Anglican program is derived by executing it in a forward manner, drawing from `sample` statements when encountered, and keeping track of density components originating from both the `sample` and `observe` terms. Specifically, let $\{x_i\}_{i=1}^{n_x} = x_1, \ldots, x_{n_x}$ represent the random variables generated from the encountered `sample` statements, where the $i$-th encountered `sample` statement has a lexical program address $a_i$, an input $\eta_i$, and a density $f_{a_i}(x_i|\eta_i)$. Analogously, let $\{y_j\}_{j=1}^{n_y} = y_1, \ldots, y_{n_y}$ represent the observed values of the $n_y$ encountered `observe` statements, which have lexical addresses $b_j$ and corresponding densities $g_{b_j}(y_j|\phi_j)$, where $\phi_j$ is analogous to $\eta_i$. The program density is now given by $\pi(x) = \gamma(x)/Z$ where

$$\gamma(x) := \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta_i) \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi_j), \tag{2}$$

$$Z := \int \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta_i) \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi_j) dx_{1:n_x}, \tag{3}$$

and the associated reference measure is implicitly defined through the encountered `sample` statements. Note here that everything (i.e. $n_x$, $n_y$, $x_{1:n_x}$, $y_{1:n_y}$, $a_{1:n_x}$, $b_{1:n_y}$, $\eta_{1:n_x}$, and $\phi_{1:n_y}$) is a random variable, but each is deterministically calculable given $x_{1:n_x}$. See Rainforth (2017, §4.3.2) for a more detailed introduction.

We denote an ***execution trace*** (i.e. realization) of an Anglican program by the sequence of the addresses of `sample` statements and the corresponding variables, namely $[a_i, x_i]_{i=1}^{n_x}$. For clarity, we refer to the sequence $a_{1:n_x}$ as the ***path*** of a trace and $x_{1:n_x}$ as the ***draws***. A program with *stochastic support* can now be more formally defined as one for which the path $a_{1:n_x}$ varies between different realizations: a different value for the path corresponds to a different *configuration* of variables being sampled.

### A.2. Density of a Path

Dividing a program into its constituent SLPs implicitly partitions the overall target density into disjoint regions, with each part defining a sub-model on the corresponding sub-space. The unnormalized density $\gamma_k(x)$ of the straight-line program $A_k$ is defined with respect to the variables $\{x_i\}_{i=1}^{n_{x,k}}$ that are paired with the addresses $\{a_i\}_{i=1}^{n_{x,k}}$ of $A_k$. We use $\mathcal{X}_k$ to denote the corresponding sub-space of $x_{1:n_{x,k}}$. Note that the union of $\mathcal{X}_k$ for all $k$ is the entire latent space defined by the overall program, $\mathcal{X} = \bigcup_{k=1}^{K} \mathcal{X}_k$. Analogously to (2),

we now have that the density of SLP $k$ is $\pi_k(x) = \gamma_k(x)/Z_k$ where

$$\begin{aligned}\gamma_k(x) &:= \gamma(x)\mathbb{I}[x \in \mathcal{X}_k] \\ &= \mathbb{I}[x \in \mathcal{X}_k] \prod_{i=1}^{n_{x,k}} f_{A_k[i]}(x_i|\eta_i) \prod_{j=1}^{n_{y,k}} g_{b_j}(y_j|\phi_j),\end{aligned} \tag{4}$$

$$Z_k := \int_{x \in \mathcal{X}_k} \gamma_k(x) dx. \tag{5}$$

Unlike for (2), $n_{x,k}$ and $A_k$ are now, critically, deterministic variables so that the support of the problem is fixed. Though $b_j$ and $n_{y,k}$ may still be stochastic, these do not effect the reference measure of the program (see Rainforth (2017, §4.4.3)) and so this does not cause a problem when trying to perform MCMC sampling.

## Appendix B. Inference Algorithms Accommodating Stochastic Support

Designing inference algorithms for models with stochastic support is typically very challenging. Some basic inference schemes, such as importance sampling from the prior, can be directly applied, but their performance deteriorates rapidly as the dimension of the model increases. Particle based inference methods such as Sequential Monte Carlo (SMC) (Wood et al., 2014; Doucet et al., 2001) can offer improvements for models with natural sequential structure, but similarly rapidly succumb to the curse of dimensionality in the majority of cases. Variational approaches, on the other hand, are typically ill-suited to this setting: though some strategies have been proposed in Paige (2016), they require substantial approximations to be made and are again only applicable to very simple problems due to difficulties with gradient estimation.

Markov chain Monte Carlo (MCMC) methods (Metropolis and Ulam, 1949) have the potential to tackle more difficult problems. In particular, reversible jump Markov chain Monte Carlo (RJMCMC) (Green, 1995, 2003) methods allow one to perform MCMC on problems with stochastic support by introducing proposals capable of transitioning between configurations. However, their application is fundamentally challenging due to the difficulty in designing proposals which can transition efficiently. Namely, proposing changes in the variable configuration introduces new variables that are not present in the current sample. Further, the posterior on the other variables may shift substantially. Consequently, one loses a notion of locality when switching configurations; having a sample in a high density region of one configuration typically provides little information about which regions have a high density for another configuration. In turn, this means that it is extremely difficult to design proposals which both efficiently move between configurations and maintain a high acceptance rate; once in a high density region of one configuration, it becomes extremely difficult to switch to another configuration. This is then compounded by the fact that RJMCMC only estimates the relative mass of each configuration through the relative frequency of transitions, giving a very slow mixing rate for the overall sampler.

The difficulty in applying RJMCMC is exacerbated in universal PPSs due to the desire to construct proposals in an automated fashion. Thus, though RJMCMC has recently been applied in the PPS context by Roberts et al. (2019), they rely on manual specification of the proposal by the user, thereby losing most of the automation that forms a core part of the

motivation for PPSs in the first place. Moreover, for many programs, it will be impractically difficult to even hand-design such a proposal.

One MCMC method that can be fully automated for PPSs is the Lightweight Metropolis Hastings algorithm (LMH) of Wingate et al. (2011) and its extensions (Ritchie et al., 2016; Tolpin et al., 2015), for which implementations are provides in a number of systems such as Venture (Mansinghka et al., 2014), WebPPL (Goodman and Stuhlmüller, 2014), and Anglican (Wood et al., 2014). LMH is based around a Metropolis-within-Gibbs (MwG) approach (Brooks et al., 2011) whereby one first samples a variable in the execution trace, $k \in 1 : n_x$, uniformly at random and then proposes a MwG transition to this sample, $x_k \to x'_k$. Unlike in a standard MwG scheme, one must further now check if this transition influences the downstream control flow of the program: we must check that the transition does not cause the downstream path to change, i.e. that we have $a'_{k+1:n'_x} = a_{k+1:n_x}$. When the path remains the same, we can reuse the downstream draws $x_{k+1:n_x}$ and, in turn, a standard MwG accept-reject step. However, when the path changes, the downstream draws no longer produce a valid execution trace. To account for this, the remainder of the trace is instead redrawn afresh by simulating from the prior, such that the proposed trace is instead $[\{a_{1:k}, a'_{k+1:n'_x}\}, \{x_{1:k-1}, x'_k, x'_{k+1:n'_x}\}]$, where $[a'_{k+1:n'_x}, x'_{k+1:n'_x}]$ is the new partial execution trace generating by this redrawing. This new sample is now accepted or rejected in the standard manner, except for an additional $n_x/n'_x$ term in the acceptance ratio.

Though widely applicable, LMH relies on proposing from the prior whenever the configuration changes for the downstream variables. This inevitably forms a highly inefficient proposal (akin to importance sampling from the prior), such that LMH typically performs very poorly for programs with stochastic support, particularly in high dimensions.

## Appendix C. Details of DCC

We now outline a particular realization of our DCC framework that we have implemented for Anglican, which can be used to perform inference in an automated fashion for any input program of Anglican. For this, we suggest particular strategies for the individual components left unspecified in the last section, emphasizing that these are not the only possible choices. Specifically, we will propose the use of a combination of Metropolis-with-Gibbs (MwG) and the parallel interacting Markov adaptive importance sampling (PI-MAIS) algorithm of Martino et al. (2017) for performing the local inference, a dynamic model discovery approach for establishing the SLPs, and a resource allocation approach based on the exploration-exploration strategy introduced in Rainforth et al. (2018).

### C.1. Local Estimators

Recall that the goal for the local inference is to estimate the local target density $\pi_k(x)$ (where we only have access to $\gamma_k(x)$), and the local marginal likelihood $Z_k$. Straightforward choices for this include (self-normalized) importance sampling and SMC as both return a marginal likelihood estimate $\hat{Z}_k$. However, knowing good proposals for these a priori is challenging and, as we discussed in §B, naïve choices like sampling from the prior are unlikely to perform well.

Thankfully, each SLP has a fixed support, which means many of complications that make inference challenging for universal PPSs no longer apply. In particular, we can use

conventional MCMC samplers—such as MH, HMC, or MwG—to approximate $\pi_k(x)$. Due to a combination of restrictions from our underlying PPS and the fact that individual variable types may be unknown or not even fixed, we have elected to use MwG in our implementation, but note that more powerful inference approaches like HMC may be preferable when they can be safely applied. To encourage sample diversity and assist in estimating $Z_k$ (see below), we further run $N$ independent MwG samplers for each SLP.

As MCMC samplers do not directly provide an estimate for the marginal likelihood, we must introduce a further estimator for $Z_k$. For this, we use the PI-MAIS approach of Martino et al. (2017). Though ostensibly an adaptive importance sampling algorithm, PI-MAIS is based around using a set of $N$ proposals each centered on the outputs of an MCMC chain. We can thus also think of it as a method for generating marginal likelihood estimates from a set of MCMC chains, which is what we require.

To be more precise, given a series of samples, $\hat{x}_{k,1:T,1:N}$, from $N$ MwG chains run for $T$ iterations each on the SLP $A_k$, for each iteration of the chain PI-MAIS introduces a mixture proposal distribution by using the combination of separate proposals (e.g. a Gaussian) centered on each of these chains:

$$q_{k,t}(\cdot|\hat{x}_{k,t,1:N}) := \frac{1}{N}\sum_{n=1}^{N} q_{k,t,n}(\cdot|\hat{x}_{k,t,n}),\ t \in \{1:T\} \tag{6}$$

This can then be used to produce an importance sampling estimate for the target, with Rao-Blackwellization typically applied across the mixture components, such that one draws $M$ samples separately from each $q_{k,t,n}$, rather than $NM$ samples from $q_{k,t}$. By proxy, this also produces a marginal likelihood estimate $\hat{Z}_k$, which is equal to the empirical average of the importance weights, where this average is taken of $N$, $T$, and $M$.

An interesting point of note is that one can use either the originally MCMC samples, or the importance samples generated by the PI-MAIS for the estimate $\hat{\pi}_k(x)$. The relative meric of these approaches depends on the exact problem (we will use the latter in our experiments). For problems where the PI-MAIS forms an efficient adaptive importance sampler, the estimate it produces will typically be preferable. However, in some cases, particularly high-dimensional problems, this sampler may struggle, so that it is more effective to take the original MCMC samples. Though it might seem that we are doomed to fail anyway in such situations, as the struggling of the PI-MAIS estimator is likely to indicate our $Z_k$ estimates are poor, this is certainly not always the case. In particular, for many problems, one SLP will dominate, i.e. $Z_{k^*} \gg Z_{k\neq k^*}$ for some $k^*$. Here we do not necessarily need an accurate estimates of the $Z_k$ to achieve an overall good approximation of the posterior, we need only identify the dominant $Z_k$.

### C.2. Extracting the SLPs

To divide a given model into its constituent sub-models expressed by SLPs, we need a mechanism for discovering these sub-models automatically.

One possible approach (Chaganty et al., 2013; Nori et al., 2014) would be to analyze the source code of the program defining the model using a static analysis, thereby extracting the set of possible execution paths of the program at compilation time. However, this is a difficult, and potentially impossible, feat to achieve for all possible programs in a universal

PPS. In particular, it fails to deal with cases where the number of the sub-models is countably infinite.

Because of these issues, we take an alternative approach based on discovering models dynamically at run time. Not only does this circumvent the need for a complex static program analysis, in settings where the number of potential models is too large to tractably enumerate, it further provides a natural approach to ensuring we only investigate models with a high potential to make a significant contribution to the overall density.

Our approach starts by executing the program forward for $T_0$ iterations to generate sample execution traces. This corresponds to drawing samples from the prior of the model. The paths traversed by these sampled traces are recorded, and our set of SLPs is initialized as that of these recorded paths. At subsequent iterations, after each local inference stage, we then perform one *global* LMH proposal based on the sub-model $A_{k^*}$ that was chosen to run local inference on, generating a new possible path $A_{k'}$. If $A_{k'}$ corresponds to an existing SLP, this sample is simply discarded. However, if it corresponds to an unseen path, it is added to our stack of models as a new SLP. To avoid the rate of models being generated outstripping our ability to perform inference on current models, this new SLP is not considered a candidate for the resource allocation (as per the next section) until some threshold for the number of times it has been proposed is reached. This also provides a mechanism for providing distinct starting points for the $N$ MCMC changes that will be run.

We note that in cases where there is a small number of discrete draws, it can sometimes be beneficial to partition our SLPs further into separate models for distinct values of these discrete variables to aid the mixing of the local MCMC sampler.

### C.3. Resource Allocation

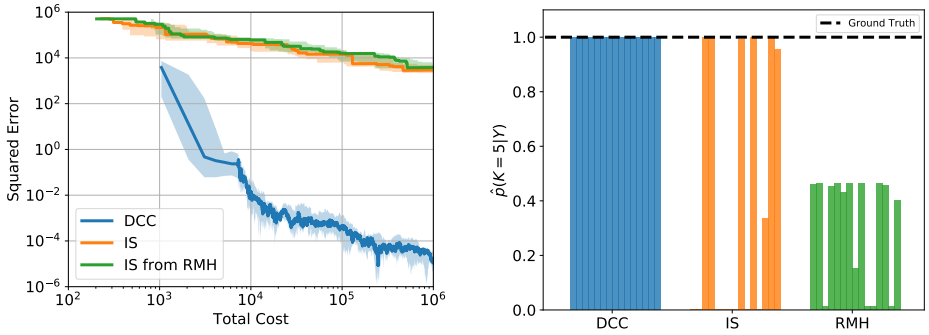
Given this dynamic set of candidate SLPs, we must now, at each iteration, choose a SLP to perform local inference on. Though valid, it is not wise to evenly split our computational resources evenly among all SLPs; it is more important to ensure we have accurate estimates for SLPs with large $Z_k$. To address this, we introduce a resource allocation scheme, based on Rainforth et al. (2018).

The resource allocation scheme is based on an Upper Confidence Bound (UCB) scheme (Carpentier et al., 2015). Specifically, at each iteration we will update the estimate for the SLP which has the largest utility, defined as

$$U_k := \frac{1}{L_k}\left((1-\delta)\bar{\tau}_k + \delta\hat{p}_j + \beta\frac{\log\sum_k L_k}{\sqrt{L_k}}\right) \tag{7}$$

where $L_k$ is the number of times DCC has performed local inference on $A_k$, $\tau_k$ is the "exploitation target" of $A_k$ (explained below) and $\bar{\tau}_k = \tau_k/\max\{\tau_{1:K}\}$, $\hat{p}_j$ is a target exploration term (explained below), and $\delta$ and $\beta$ are hyper-parameters, adapted from Rainforth et al. (2018, Eq. 6 in §5).

As proved in Rainforth et al. (2018, §5.1), the optimal asymptotic allocation strategy is to choose each $A_k$ in proportion to $\tau_k = \sqrt{Z_k^2 + (1+\kappa)\sigma_k^2}$ where $\kappa$ is a smoothness hyper-parameter, $Z_k$ is the local marginal likelihood, and $\sigma_k^2$ is the variance of the weights of the individual samples used to generate $Z_k$. Intuitively, this allocates resources not only




(a) Squared error $||\log \hat{Z} - \log Z_{true}||^2$

(b) Estimates $\hat{p}(K = 5 \,|\, y_{1:N_y})$ of 15 runs.

Figure 4: We compare DCC (ours) against IS and RMH on the convergence of the log marginal likelihood (4(*a*)) and the posterior distribution of the number of the clusters (4(*b*)) over 15 independent runs. The ground truth of the log marginal and the posterior probability $p(K = 5 \,|\, y_{1:N_y})$ were estimated using a large number of samples with a manually adapted proposal. In Figure 4(*a*), we show the squared error of the log marginal likelihood estimates with the solid line being the median and the shading region $25\% - 75\%$ quantiles. In Figure 4(*b*), we report the estimated posterior probability of $K = 5$ of each run, where the true estimate is around 0.9998. We see that DCC substantially outperforms the baselines for both.

to the SLPs with high marginal probability mass, but also to the ones having high variance on our estimate of it. We normalize $\tau_k$ by the maximum of $\tau_{1:K}$ as the reward function in UCB is usually in $[0, 1]$.

The target exploration term $\hat{p}_j$ is a subjective tail-probability estimate on how much the local inference *could improve* in estimating the local marginal likelihood if given more computations. This is motivated by the fact that estimating $Z_k$ accurately is difficult, especially at the early stage of inference. One might miss substantial modes if only relying on optimism boost to undertake exploration. As per Rainforth et al. (2018), we realize this insight by extracting additional information from the log weights. Namely, we define $\hat{p}_k := P(\hat{w}_k(T_a) > w_{th}) \approx 1 - \Psi_k(\log w_{th})^{T_a}$, which means the probability of obtaining at least one sample with weight $w$ that exceeds some threshold weight $w_{th}$ if provided with $T_a$ "look-ahead" samples. Here $\Psi_k(\cdot)$ is a cumulative density estimator of the log local weights, $T_a$ is a hyperparameter, and $w_{th}$ can be set to the maximum weight so far among all SLPs. If $\hat{p}_k$ is high, it implies that there is a high chance that one can produce higher estimates of $Z_k$ given more budget.

## Appendix D. Experiment Details

### D.1. Gaussian Mixture Model

We first consider a Gaussian Mixture Model where the number of clusters is unknown. Specifically, we have

$$\begin{aligned} K &\sim \text{Uniform}\{K_{min}, K_{min}{+}1, \ldots, K_{max}\} \\ \mu_k | K &\sim \mathcal{U}\left(a + \frac{(b-a)(k-1)}{K}, a + \frac{(b-a)(k)}{K}\right) \quad \text{for } k = 1 \ldots K, \\ z_n | K &\sim \text{Cat}(\{1/K, ..., 1/K\}) \quad \text{for } n = 1 \ldots N_y, \\ y_n | (z_n{=}k, \mu_k) &\sim \mathcal{N}(\mu_k, \sigma_k) \quad \text{for } n = 1 \ldots N_y. \end{aligned}$$

Here $K$ is the number of clusters, $\mu_{1:K}$ are the cluster centers, $z_{1:N_y}$ are the cluster assignments, $y_{1:N_y}$ is the observed data, and all other terms are fixed parameters. When conducting inference, we can analytically marginalize out the cluster assignments $z_{1:N_y}$ and perform inference on $K$ and $\mu_{1:K}$ only.

To benchmark DCC, we generated a synthetic dataset of $y_{1:150}$ for an one-dimensional mixture of five clusters by setting $K_{min} = 2$, $K_{max} = 8$, $a = 0$, $b = 20$ and $\sigma_{1:K_{max}} = 0.1$. Note that there are seven sub-models for this dataset. We compare the performance of our DCC method against two baselines: Anglican's importance sampling (IS) and RMH (Le, 2015) (a variant of LMH) implementations, taking the same computational budget (one million samples in total).

We first examine the convergence of the overall log marginal likelihood $Z$. As RMH cannot be used directly here, we instead draw importance samples centered around the RMH chain in a manner akin to PI-MAIS. Figure 4($a$) shows that DCC outperforms both baselines by many orders of magnitude.

To further investigate the source of these improvements, we further look into the posterior distribution of $K$ and report the estimates of $\hat{p}(K = 5 \,|\, y_{1:N_y})$ in Figure 4($b$). We see that IS occasionally gives reasonable estimates (6 out of 15 runs) but the performance is unstable. This is due to the fact that the posterior mass mostly concentrated within one sub-model ($K = 5$), and further highly peaked within that model. The IS scheme evenly spreads computation resources among the different sub-models and, moreover, struggles to make effective proposals within these models.

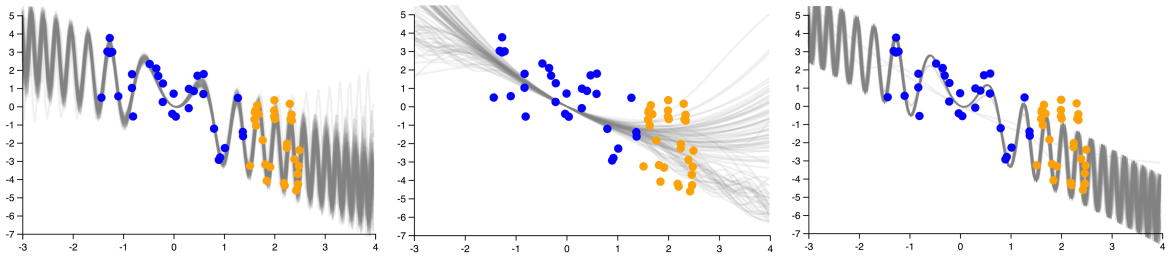
RMH, on the other hand, has a tendency to become stuck in one sub-model, and it did not accurately estimate $\hat{p}(K = 5 \,|\, y_{1:N_y})$ for any of the runs.

### D.2. Function Induction

Function induction is an important task for automated machine learning and has been investigated in many scenarios (Duvenaud et al., 2013; Kusner et al., 2017). In PPSs, it is typically tackled using a probabilistic context free grammar (PCFG) (Manning et al., 1999). We specify the structure of a candidate function using a PCFG and distribution over the function parameters, and estimate the posterior of both for given data.

Our PCFG model consists of four production rules:

$$R = \{e \to x \mid x^2 \mid \sin(a * e) \mid a * e + b * e\}$$



(*a*) DCC (ours) (*b*) IS (*c*) RMH

Figure 5: Posterior distribution $p(\Theta|D)$ estimated by DCC (ours), IS and RMH under the same computational budget. Blue points represent the observed data $D$ and orange ones the test data $D'$. Grey lines are the posterior samples of the functions from one run for the three algorithms.

| | **DCC (ours)** | IS | RMH |
|---|---|---|---|
| LML | **-14.483 ± 0.219** | -213.642 ± 0.335 | -19.870 ± 7.262 |

Table 2: Mean and one standard derivation of the MLL over 15 independent runs. (The higher, the better.)

where $x$ and $x^2$ are terminal symbols, $a$ and $b$ are unknown coefficient parameters, and $e$ is a non-terminal symbol. The rules have fixed probabilities $p_R$. The model also has the prior distributions for each parameter.

To generate a function in this model, we first sample its structure from the PCFG $R$. Next, we decide parameters in the sampled structure, by treating the parameters as all different variables and sampling them from the prior distribution. Let $\Theta$ be the collection of all the latent variables used in this generative process. That is, $\Theta$ consists of the discrete variables recording the choices of the grammar rules and the coefficients in the sampled structure. Conditioned on the training data $D$, we want to infer the posterior distribution $p(\Theta|D)$, and calculate the predictive distribution for the test data $D' = \{(x_j^*, y_j^*)\}_{j=1}^N$.

In our experiment, we control the number of sub-models by requiring that the model use the PCFG in a restricted way: a sampled function structure should have depth at most 3 and cannot use the plus rule consecutively. We generate a synthetic dataset of 30 training data points (Figure 5, blue points) and compare the performance of DCC against IS and RMH on estimating the posterior distribution and the posterior predictive under the same computational budget (one million samples in total) over 15 independent runs.

Figure 5 shows the posterior samples generated by DCC, IS and RMH over one run, with the training data $D$ marked blue and the test data $D'$ in orange. The DCC samples capture the periodicity of the training data and in general interpolates them well, while remaining uncertain in the regions of no data. This indicates good inference results on both the structure of a function (determined by the PCFG) and the coefficients of the structure. Though RMH does find some good functions, it becomes stuck in a particular mode and doe not fully capture the uncertainty in the model, leading to poor predictive performance.

Table 2 shows the test log marginal likelihood (LML) estimates of the three algorithms, i.e. LML $:= \log \sum_{j=1}^{N} \int_{\Theta} p(y_j^*|x_j^*, \Theta) p(\Theta|D) d\Theta$. The LML measures how likely the predicted $\hat{y}_j^*$ on each test $x_j^*$ is to be the true $y_j^*$ in log scale. We compared the LML for the three algorithms over 15 independent runs. DCC clearly outperforms the baselines both in terms of predictive accuracy and stability. The samples from IS approximate the posterior badly so unsurprisingly its LML is low. RMH has a LML "close" to DCC, though the probability is 200 smaller in non-log space. A more substantial problem in RMH is its high variance of the LML. This is caused by it struggling to move and the results from runs to runs vary significantly.

To test the effectiveness of the resource allocation strategy of DCC, we also compare computational resource spent for each sub-model $A_k$ with the convergence of the local marginal likelihood estimate $\log \hat{Z}_k$ of $A_k$. Our comparison is shown in Figure 6, which implies that DCC indeed spends more computational resource on sub-models with high probability mass, while also exploring the other sub-models occasionally. For our training data $D$, four sub-models (out of 26) contain most of the probability mass. Two of them (models 15, 18) are functions of the form $f(x) = a_1 x + a_2 \sin(a_3 x^2)$ modulo symmetry, which is used to generate $D$. The other two sub-models (models 23,24) are functions having the form $f(x) = a_1 \sin(a_2 x) + a_3 \sin(a_4 x^2)$, which can also match the data well in the region of the training data $(-1.5, 1.5)$ (under appropriately chosen $a_i$'s).

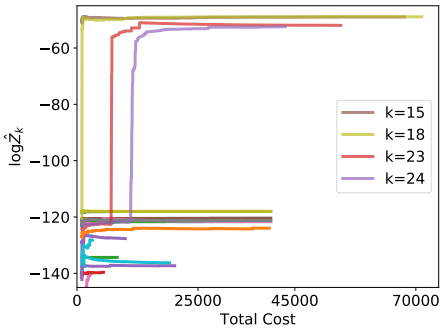



Figure 6: DCC's estimate of the local log marginal likelihood $\log Z_k$ for each sub-model.