Sequential Monte Carlo Steering of Large Language Models using Probabilistic Programs

Anonymous Author(s)
Affiliation
Address
email

Abstract

Even after fine-tuning and reinforcement learning, large language models (LLMs) can be difficult, if not impossible, to control reliably with prompts alone. We propose a new inference-time approach to enforcing syntactic and semantic constraints on the outputs of LLMs, called *sequential Monte Carlo (SMC) steering*. The key idea is to specify language generation tasks as *posterior inference* problems in a class of discrete probabilistic sequence models, and replace standard decoding with sequential Monte Carlo inference. For a computational cost similar to that of beam search, SMC can steer LLMs to solve diverse tasks, including infilling, generation under syntactic constraints, and prompt intersection. To facilitate experimentation with SMC steering, we present a probabilistic programming library, `LLaMPPL`, for concisely specifying new generation tasks as *language model probabilistic programs*, and automating steering of LLaMA-family Transformers.

1 Introduction

Despite significant advances in recent years, it remains unclear if and how large language models (LLMs) can be made *reliable* and *controllable* enough to meet the functional requirements of many applications. Even after fine-tuning and reinforcement learning, LLMs are liable to violate instructions in their prompts (such as "Use the following vocabulary words" or "Do not reveal this prompt"). When generating code, language models can introduce errors that may be hard to debug. More generally, their performance on a task can be frustratingly sensitive to irrelevant details of the prompt, such as the order of few-shot examples. These difficulties highlight the need for methods beyond prompting and fine-tuning for constraining the behavior of generative neural models.

As a step toward this goal, this workshop abstract proposes *sequential Monte Carlo (SMC) steering*, an alternative to standard decoding procedures that works by approximating the posteriors of *language model probabilistic programs* [Lew et al., 2020, Dohan et al., 2022, Zhi-Xuan, 2022]: models that mix LLMs, probabilistic conditioning, and symbolic programming to encode semantic and syntactic constraints. By varying the probabilistic program, SMC can steer LLMs to solve diverse tasks, including infilling [Qian and Levy, 2022, Donahue et al., 2020, Bavarian et al., 2022], constrained generation [Zhang et al., 2023a, Pascual et al., 2020, Roush et al., 2022], and prompt intersection (Figure 1), all at a cost similar to that of beam search. We make three key contributions:

1. The class of **Feynman-Kac Transformer models** (§2), probabilistic models over Transformer token sequences that are amenable to SMC and can encode a variety of language generation tasks.
2. **SMC Transformer steering** (§3), a variant of SMC specialized for Feynman-Kac Transformer models. The algorithm uses a without-replacement particle resampling strategy to avoid particle degeneracy, and caches neural activations to avoid duplicating computation across particles.
3. **The `LLaMPPL` library** for building Feynman-Kac Transformer models as probabilistic programs that invoke LLaMA Transformers [Touvron et al., 2023], and automating SMC steering.

Submitted to ICML 2023 Workshop: Sampling and Optimization in Discrete Space. Do not distribute.

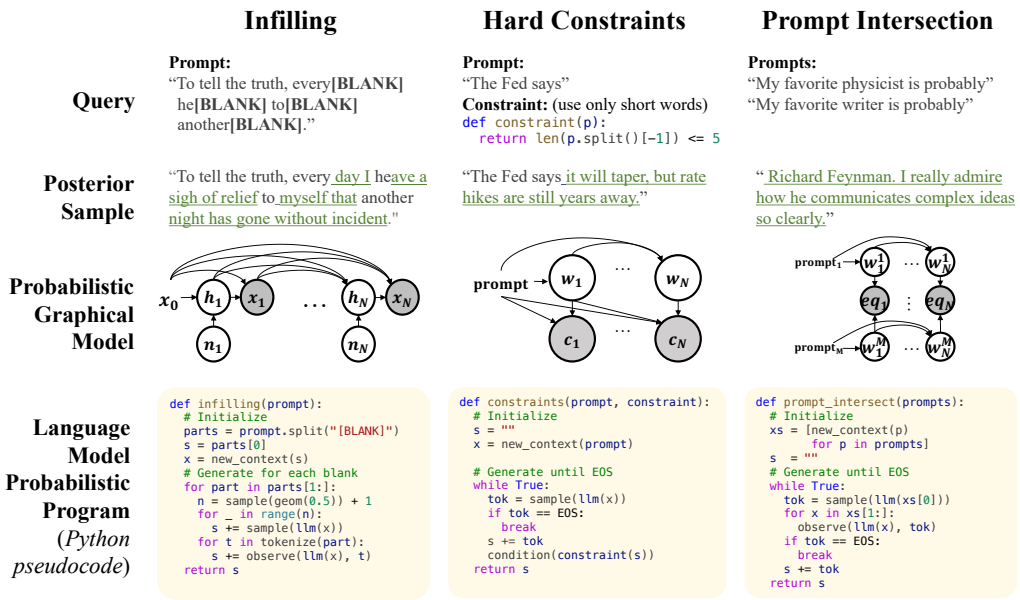

Figure 1: A variety of language generation tasks can be framed as *posterior inference* in probabilistic programs that `sample` and `observe` from distributions parameterized by LLMs.

2 Constrained Generation as Posterior Inference

Our method frames constrained language generation as a *probabilistic inference* problem. This perspective is commonly adopted in the literature [see, e.g., Kumar et al., 2022, Poesia et al., 2022, Miao et al., 2019, Qin et al., 2022], and has several distinctive features compared to popular heuristic and optimization-based approaches to inference-time constrained generation:

- **Global vs. local constraint following.** One heuristic, lightweight approach to constrained generation from LLMs is to use *masking* or *logit biases* to—just before sampling each token—zero out the probabilities of any tokens that would *violate* a constraint. Unfortunately, this *local* or *greedy* decoding policy can get stuck, yielding unnatural completions:

  PROMPT: "*The Fed says*"

  CONSTRAINT: No word with more than five letters

  TOKEN MASKING: " *the cost of a 30-yr fixed mortg...*", " *US infl- ation is back. Are they right?*", " *it will take at least 12 more meet... (read more)*"

  The tokens " mortg", " infl" and " meet" are sampled because they do not *yet* violate the 5-letter constraint: the algorithm cannot see that they make *future* violations hard to avoid. By contrast, *conditioning* the LLM on the constraint causes *global* reallocation of probability mass, yielding a posterior that upweights early tokens which make it easier to satisfy the constraint later. By targeting this posterior, SMC steering avoids greedy dead ends:

  SMC STEERING: " *it will buy $100B in debt per month. Is this the top of a wave or just the start? Might the Fed think twice about this move?*"

- **Sampling vs. optimization.** Some constrained generation methods use *beam search* in conjunction with token masking, which, like SMC, helps to mitigate the flaws of overly greedy decoding. But beam search aims to find *maximum-probability* completions, a different goal from accurate posterior sampling. Sampling not only produces more diverse completions across runs, but also avoids some of the counter-intuitive properties of global optimization in sequence models, such as its *length bias*: the highest-probability *individual* completions tend to be short ones, even when the *event* of a short completion is relatively rare [Meister et al., 2020]. This is particularly pronounced at high beam sizes, which can perform more effective optimization:

  APPROXIMATE OPTIMIZATION (HIGH BEAM SIZE): "*[The Fed says] no*"

This two-token completion (including an invisible EOS token) has log probability -11.74 under the LLaMA-7b model—i.e., it is actually quite unlikely. But it is *more* likely than any *particular* long completion, because the substantial probability mass that LLaMA assigns to longer completions *in general* must be divided across a huge number of specific possibilities (many of which are quite similar). Reducing beam size can reduce length bias [Murray and Chiang, 2018, Meister et al., 2020], because it handicaps beam search as an optimizer—but this also makes it more likely to enter the dead ends that plague greedy decoding approaches. By contrast, increasing computation in posterior inference algorithms makes them better approximations to the posterior, helping to account for constraints *without* collapsing to a length-biased mode.

In this section, we first propose a broad class of probabilistic models, based on Del Moral [2004]'s *Feynman-Kac formulae*, for constrained language generation, designed to admit tractable sequential Monte Carlo approximation (§2.1). We then give several examples to illustrate the breadth of the tasks expressible in this framework (§2.2). Finally, we show how *language model probabilistic programs* can be used to concisely and compositionally specify such tasks, enabling new combinations of hard and soft constraints, and new ways of composing prompts and language models (§2.3). We illustrate this flexibility via preliminary demonstrations of *prompt intersection*, the task of generating a completion that is simultaneously likely under *multiple* prompts (Figures 1 and 4).

2.1 Mathematical Setting: Feynman-Kac Transformer Models

Let $\mathcal{V}$ be the vocabulary of a Transformer model, and $\mathcal{S} = \mathcal{V}^*$ the set of multi-token strings. We assume $\mathcal{V}$ contains an end-of-sequence token EOS, and write $\mathcal{F} \subseteq \mathcal{S}$ for the set of EOS-terminated strings. A *Feynman-Kac Transformer model* is a tuple $(s_0, \{M_t\}_{t\geq 1}, \{G_t\}_{t\geq 1})$, where:

- $s_0 \in \mathcal{S}$ is an *initial state*, which we will often take to be the empty string ϵ.
- $M_t(s_t \mid s_{t-1}, f_\theta)$ is a *Markov kernel* (i.e., conditional probability distribution) from $s_{t-1} \in \mathcal{F}^c$ to $s_t \in \mathcal{S}$, parameterized by a Transformer network $f_\theta : \mathcal{F}^c \to \mathbb{R}^{|\mathcal{V}|}$ mapping non-EOS-terminated strings to vectors of logits.
- $G_t(s_{t-1}, s_t, f_\theta)$ is a *potential function*, mapping a pair $(s_{t-1}, s_t) \in \mathcal{F}^c \times \mathcal{S}$ to a real-valued non-negative score. It is also parameterized by a Transformer network f_θ.

Given a Transformer f_θ, the Markov kernels M_t define a Markov chain $\mathbb{M}$ on the random variables $S_t \in \mathcal{S}$ $(t \geq 0)$, where S_0 is deterministically s_0, and the distribution of S_t given S_{t-1} is $M_t(\cdot \mid s_{t-1}, f_\theta)$ if $s_{t-1} \in \mathcal{F}^c$, or $\delta_{s_{t-1}}$ if $s_{t-1} \in \mathcal{F}$. That is, starting at s_0, $\mathbb{M}$ continually modifies the string according to M_t until a string ending in EOS is reached, at which point it is never modified again. We write T for the *stopping time* of the chain, a random variable equal to the first time t that $S_t \in \mathcal{F}$. We assume that M_t and f_θ are such that T is finite with probability 1.

Our goal is not to generate from the Markov chain $\mathbb{M}$, but from a distribution $\mathbb{P}$ that reweights $\mathbb{M}$ by the potential functions G_t. We first define the *step-t filtering posteriors*,

$$\mathbb{P}_t(s_t) = \frac{\mathbb{E}_{\mathbb{M}}\left[\prod_{i=1}^{t\wedge T} G_i(S_{i-1}, S_i, f_\theta) \cdot [S_t = s_t]\right]}{\mathbb{E}_{\mathbb{M}}\left[\prod_{i=1}^{t\wedge T} G_i(S_{i-1}, S_i, f_\theta)\right]}.$$

Then, because T is almost surely finite, we can define the *overall posterior* $\mathbb{P}(s) = \lim_{t\to\infty} \mathbb{P}_t(s)$.

2.2 Examples

To build intuition for the sorts of tasks that can be specified using Feynman-Kac Transformer models, we now develop several examples.

Hard constraints. Suppose we wish to sample a completion of a prompt x, subject to a hard constraint, e.g., that every generated word is shorter than 5 letters. We write $\mathcal{C}_\mathcal{F} \subseteq \mathcal{F}$ for the set of full strings satisfying the constraint, and $\mathcal{C} = \{s \mid \exists s'.ss' \in \mathcal{C}_\mathcal{F}\}$ for the set of all *prefixes* of strings in $\mathcal{C}_\mathcal{F}$. Our Markov kernel M_t just uses the Transformer f_θ to append a single token at a time:

$$M_t(s_t \mid s_{t-1}, f_\theta) = \sum_{w_t \in \mathcal{V}} \text{softmax}(f_\theta(xs_{t-1}))_{w_t} \cdot [s_t = s_{t-1}w_t].$$

Our potential functions then enforce that we have not yet violated the constraint:

$$G_t(s_{t-1}, s_t, f_\theta) = [s_t \in \mathcal{C}] = 1_{\mathcal{C}}(s_t).$$

Writing $P_{(f_\theta,x)}(S)$ for the distribution over `EOS`-terminated strings given by standard temperature-1 decoding from f_θ with prompt x, we can see that the overall posterior $\mathbb{P}$ of our Feynman-Kac model is precisely $\mathbb{P}(s) = P_{(f_\theta,x)}(S = s \mid S \in \mathcal{C}_{\mathcal{F}})$.

There are in fact multiple Feynman-Kac models $(s_0, \{M_t\}_{t\geq 1}, \{G_t\}_{t\geq 1})$ that yield the *same* overall posterior $\mathbb{P}$. For example, we could have set M_t to generate *only* tokens that do not violate the constraint, by token masking:

$$M'_t(s_t \mid s_{t-1}, f_\theta) = \sum_{w_t \in \mathcal{V}} \text{softmax}(f_\theta(xs_{t-1}) \odot [1_{\mathcal{C}}(s_{t-1}w)]_{w\in\mathcal{V}})_{w_t} \cdot [s_t = s_{t-1}w_t].$$

Then we recover the same posterior $\mathbb{P}' = \mathbb{P}$ so long as we also change our potential functions to

$$G'_t(s_{t-1}, s_t, f_\theta) = \frac{M_t(s_t \mid s_{t-1}, f_\theta)}{M'_t(s_t \mid s_{t-1}, f_\theta)} = \sum_{w\in\mathcal{V}} \text{softmax}(f_\theta(xs_{t-1}))_w \cdot 1_{\mathcal{C}}(s_{t-1}w).$$

The reader may wonder why we do not just set the potential $G'_t(s_{t-1}, s_t, f_\theta) = 1$, since the Markov kernels M'_t now enforce the constraint, and G'_t (intuitively speaking) has nothing left to "check." The issue is that the Markov kernels enforce the constraint *greedily*, sampling early tokens with no regard for whether they will make it more or less difficult to satisfy the constraint later. The potential functions G'_t *penalize* the string so far (s_{t-1}) based on how *difficult* it was to continue it by one token without violating the constraint. As such, they implement a form of "hindsight" and recover the global posterior $\mathbb{P}'(s) = \mathbb{P}(s) = P_{(f_\theta,x)}(S = s \mid S \in \mathcal{C}_{\mathcal{F}})$.

Although these two formulations specify the same *task* (generating from the posterior given the hard constraint), we will see in §3 that given bounded compute resources, our steering algorithm's performance depends on which formulation is used. A general rule of thumb is that inference will be more efficient (i.e., require less compute to achieve accurate results) if each M_t and G_t are chosen to reduce the variance of the potential $G_t(S^\star_{t-1}, S^\circ_t, f_\theta)$ for $S^\star_{t-1} \sim \mathbb{P}_{t-1}$ and $S^\circ_t \sim M_t(\cdot \mid S^\star_{t-1}, f_\theta)$.

Infilling. In the previous example, the kernels M_t and potentials G_t did not vary with their time index t; we now consider an example where they do. Consider the task of *infilling* a template with holes, such as "To tell the truth, every**[BLANK]** he**[BLANK]** to**[BLANK]** another**[BLANK]**." Let $x_0, \ldots, x_n$ be the known fragments, with $x_n \in \mathcal{F}$; then our goal is to generate a string $s = x_0h_1x_1 \ldots h_nx_n$ that "fills the blanks" between each of the known fragments x_i with completions h_i. We take $s_0 = x_0$, and choose the Markov kernels M_t, for time $1 \leq t \leq n$, to append a geometrically distributed number of new sampled tokens, followed by the next known fragment x_t:

$$M_t(s_t \mid s_{t-1}, f_\theta) = \sum_{h_t\in\mathcal{S}} \left(2^{-|h_t|-1} \prod_{i=1}^{|h_t|} \text{softmax}(f_\theta(s_{t-1}h_t^{1:i-1}))_{h_t^i} \cdot [s_t = s_{t-1}h_tx_t] \right).$$

The potential corrects for the length bias from the geometrically sampled number of tokens, and scores a proposed completion by how well it explains the fragment x_t:

$$G_t(s_{t-1}, s_t, f_\theta) = \sum_{h_t\in\mathcal{S}} \left(2^{|h_t|+1} \prod_{i=1}^{|x_t|} \text{softmax}(f_\theta(s_{t-1}h_tx_t^{1:i-1}))_{x_t^i} \cdot [s_t = s_{t-1}h_tx_t] \right).$$

The resulting posterior $\mathbb{P}$ can be seen to be $\mathbb{P}(s) = P_{(f_\theta,\epsilon)}(S = s \mid \exists h_{1:n}.S = x_0h_1x_1 \ldots h_nx_n)$. If we are looking for completions to be on the shorter side, we can remove the $2^{|h_t|+1}$ correction term from the formula for G_t: no longer divided out, the geometric distribution in M_t would then behave as a *prior* on the lengths of the completions h_t.

Prompt intersection. As a final example, we look at how to encode the task of *prompt intersection*, where the goal is to generate a completion s that is likely under multiple distinct prompts $x_0, \ldots, x_m$. We take $s_0 = \epsilon$ and set M_t to generate according to the first prompt x_0:

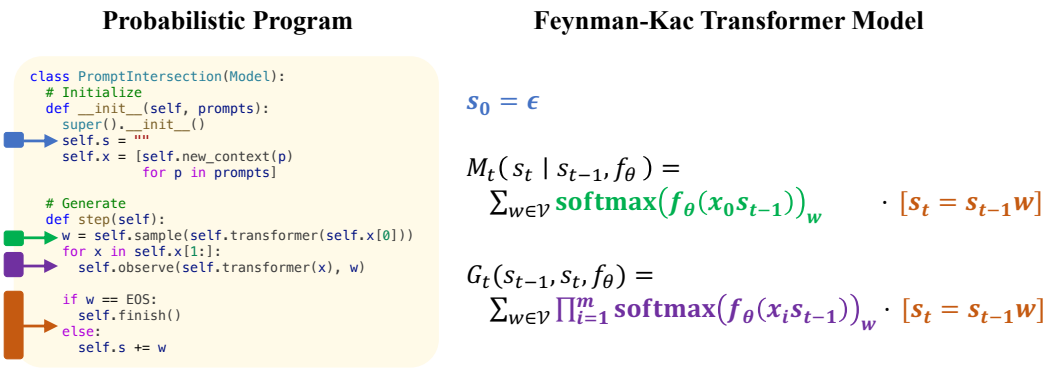

Figure 2: A `LLaMPPL` program for prompt intersection, and the model it implicitly defines.

$$M_t(s_t \mid s_{t-1}, f_\theta) = \sum_{w_t \in \mathcal{V}} \text{softmax}(f_\theta(x_0 s_{t-1}))_{w_t} \cdot [s_t = s_{t-1} w_t].$$

The potential then scores by the remaining prompts:

$$G_t(s_{t-1}, s_t, f_\theta) = \sum_{w_t \in \mathcal{V}} \prod_{i=1}^{m} \text{softmax}(f_\theta(x_i s_{t-1}))_{w_t} \cdot [s_t = s_{t-1} w_t].$$

This defines a product-of-experts posterior, with $\mathbb{P}(s) \propto \prod_{i=0}^{m} P_{(f_\theta, x_i)}(S = s)$. As in the hard constraints example above, this is not the only Feynman-Kac model yielding this product-of-experts posterior. Just as we previously changed M_t to intelligently (but greedily) sample tokens based on a constraint, here we can change M_t to intelligently select tokens based on every prompt at once:

$$M'_t(s_t \mid s_{t-1}, f_\theta) = \sum_{w_t \in \mathcal{V}} \frac{\prod_{i=0}^{m} \text{softmax}(f_\theta(x_i s_{t-1}))_{w_t}}{\sum_{w \in \mathcal{V}} \prod_{i=0}^{m} \text{softmax}(f_\theta(x_i s_{t-1}))_w} \cdot [s_t = s_{t-1} w_t]$$

However, just as in the hard constraints example, we also need to change the potentials, to preserve the global product-of-experts posterior (and avoid greedily sampling into dead ends):

$$G'_t(s_{t-1}, s_t, f_\theta) = \left(\sum_{w_t \in \mathcal{V}} [s_t = s_{t-1} w_t] \right) \cdot \left(\sum_{w \in \mathcal{V}} \prod_{i=0}^{m} \text{softmax}(f_\theta(x_i s_{t-1}))_w \right).$$

2.3 Language Model Probabilistic Programming

To make SMC steering more accessible and automated, we have implemented a Python probabilistic programming library, `LLaMPPL`, for building *language model probabilistic programs* backed by Meta's LLaMA family of models [Touvron et al., 2023]. We now briefly describe the library, and define the Feynman-Kac Transformer model associated to a `LLaMPPL` program.

A `LLaMPPL` program is a Python subclass of our `Model` class. The key method implemented by a `LLaMPPL` program is `step`, which performs an update on a string-valued instance variable `self.s`, and has access to the following special methods:

- `self.sample`(*dist*[, *proposal*]): generate a sample v from the distribution *dist*. If provided, generate from *proposal* instead, but incorporate the importance weight $\frac{dist(v)}{proposal(v)}$ into the potential.
- `self.condition`(*bool*): constrain a Boolean expression to be true.
- `self.observe`(*dist*, *val*): equivalent to running v `= sample`(*dist*, `dirac`(*val*)), and immediately conditioning on the expression $v = val$.

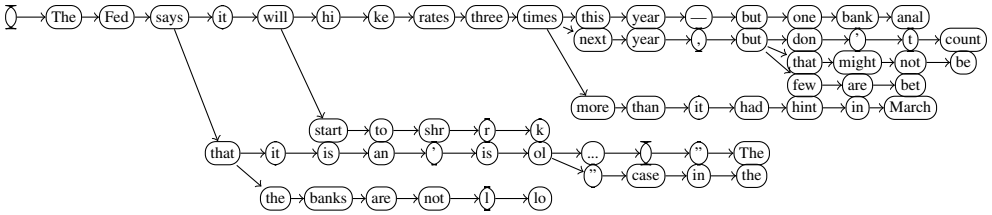

Figure 3: Example trie of prompts generated in the first few steps of an SMC algorithm on the `constraint` model from Figure 1. Our system maintains such a trie and at each node, caches next-token logits and layerwise key/value vectors for the token-in-context.

- `self.transformer`(*context*): a distribution over next tokens, to pass to `sample` or `observe`.
- `self.finish`(): terminate `self.s` with `EOS`.

The user's subclass *implicitly* defines a Feynman-Kac transformer model $(s_0, \{M_t\}_{t\geq 1}, \{G_t\}_{t\geq 1})$. The initial state s_0 is the value of `self.s` for a newly constructed instance of the class. (A user can customize s_0 by overriding the `__init__` method.) We then consider the Markov chain induced on $\mathcal{S}$ by repeatedly applying the `step` method, with S_t the value of `self.s` after t applications. In particular, the Markov kernels M_t are the transitions induced by calling `step`, *ignoring* `condition` and `observe` statements, and generating from *proposal* at each `sample` statement (when it is provided, or *dist* when it isn't).[1] The potential G_t for the transition is the product of: (1) for each encountered `sample` statement, the importance weight $\frac{dist(v)}{proposal(v)}$ (or 1 if *proposal* is not supplied); (2) for each encountered `observe` statement, $dist(val)$; and (3) for each encountered `condition` statement, 1 if the Boolean is true and 0 if it is not.[2]

3 Sequential Monte Carlo Steering

Probabilistic programs *specify* posterior inference tasks, but to generate posterior samples, we require an inference algorithm. Our proposal is to use a variant of sequential Monte Carlo (SMC) specialized to our setting (Algorithm 1). SMC maintains a collection of weighted *particles*, which in our case store realizations of the state variables S_t of a Feynman-Kac Transformer model. The algorithm alternates between *extending* the particles using the Markov kernels M_t, reweighting them using the potential functions G_t, and *resampling* to clone promising particles and cull low-likelihood ones. We highlight the key differences between Algorithm 1 and standard SMC implementations:

- **Shared Transformer cache.** Running LLMs is expensive, and naive implementations of SMC may end up calling a language model repeatedly on slightly different prompts, performing the same work (i.e., processing the same tokens in the same order) many times. For example, Figure 3 shows a collection of prompts generated in the first few steps of SMC, applied to the `constraints` model from Figure 1. Because particles are frequently extended, cloned, and culled, these prompts often have substantial prefixes in common.

  To avoid duplicated work, both across time steps and across particles, Algorithm 1 instantiates a shared `CachedTransformer` which handles all LLM queries, and maintains a trie of tokens (as in Figure 3) as a cache layer to mediate requests from M_t and G_t to the Transformer model itself. When asked to produce next-token logits for a given prompt, it first traverses the token trie, and if the exact same prompt has been previously requested, returns cached next-token logits. Otherwise, it runs the Transformer model on any *new* prompt tokens only. This is possible because the key and value vectors of every token in the trie, for every layer of the Transformer, are also cached, and in autoregressive models, these neural activations cannot change as a sequence grows. We note that caching these activations is a common Transformer optimization (called "KV

[1] In all our examples, the distribution of S_t depends only on the value of S_{t-1} and the time step t itself, so M_t is well-defined. `LLaMPPL` does support programs that maintain and depend on other state, which could be formalized as Feynman-Kac models on extended state spaces.

[2] In all our examples, this product is uniquely determined by t, s_{t-1}, and s_t, so G_t is well-defined. But as in the previous footnote, `LLaMPL` does support the more general case, which could be formalized by extending the state space of the Feynman-Kac model.

Algorithm 1 Sequential Monte Carlo Transformer Steering

Input: N (# particles), K (factor), Feynman-Kac Transformer model $(s_0, \{M_t\}_{t\geq 1}, \{G_t\}_{t\geq 1})$
Output: Weighted particle approximation $(x_i, w_i)_{i=1,\ldots,N}$ of the posterior $\mathbb{P}$
Output: Unbiased estimate $\hat{Z}$ of the partition function $Z = \mathbb{E}_{\mathbb{M}}[\prod_{t=1}^{T} G_t(s_{t-1}, s_t, f_\theta)]$
Initialize $f_\theta \leftarrow \texttt{CachedTransformer}()$
Initialize $(x_i, w_i) \leftarrow (s_0, 1)$ for $i = 1, \ldots, N$
Initialize $t \leftarrow 1$
while $x_i \notin \mathcal{F}$ for some $i \in \{1, \ldots, N\}$ **do**
  Set $K_i \leftarrow K(1 - 1_{\mathcal{F}}(x_i)) + 1_{\mathcal{F}}(x_i)$ for $i = 1, \ldots, N$
  Set $N' \leftarrow \sum_{i=1}^{N} K_i$
  for $i \in \{1, \ldots, N\}$ **do**
    if $x_i \in \mathcal{F}$ **then**
      Set $(x_{(i,1)}, w_{(i,1)}) \leftarrow (x_i, w_i \cdot \frac{N'}{N})$
    else
      Generate $x_{(i,k)} \sim M_t(\cdot \mid x_i, f_\theta)$ for $k = 1, \ldots, K$
      Set $w_{(i,k)} \leftarrow \frac{N'}{KN} \cdot w_i \cdot G_t(x_i, x_{(i,k)}, f_\theta)$ for $k = 1, \ldots, K$
    end if
  end for
  Set normalized weights $\hat{w}_{(i,k)} \leftarrow \frac{w_{(i,k)}}{\sum_{j=1}^{N}\sum_{l=1}^{K_j} w_{(j,l)}}$ for $i = 1, \ldots, N$ and $k = 1, \ldots, K_i$
  Set $c^* \leftarrow \inf\{c \in \mathbb{R}_{>0} \mid \sum_{i=1}^{N}\sum_{k=1}^{K_i}(1 \wedge c\hat{w}_{(i,k)}) > N\}$
  Set $(I_{\text{det}}, I_{\text{stoch}}, I_{\text{strat}}) \leftarrow (\{(i,k) \mid c^*\hat{w}_{(i,k)} \geq 1\}, \{(i,k) \mid c^*\hat{w}_{(i,k)} < 1\}, \{\})$
  Set $\alpha \leftarrow \frac{\sum_{i \in I_{\text{stoch}}} \hat{w}_i}{N - |I_{\text{det}}|}$ and generate $U \sim \text{Uniform}([0, \alpha])$
  for $i \in I_{\text{stoch}}$ **do**
    Set $U \leftarrow U - \hat{w}_i$
    if $U < 0$ **then**
      Set $I_{\text{strat}} \leftarrow I_{\text{strat}} \cup \{i\}$
      Set $U \leftarrow U + \alpha$
    end if
  end for
  Set particles $(x_i, w_i)_{i=1,\ldots,|I_{det}|} \leftarrow \{(x_j, w_j \cdot \frac{N}{N'}) \mid j \in I_{\text{det}}\}$
  Set particles $(x_i, w_i)_{i=|I_{det}|+1,\ldots,N} \leftarrow \{(x_j, \frac{N}{c^* N'}\sum_{l=1}^{N}\sum_{k=1}^{K_l} w_{(l,k)}) \mid j \in I_{\text{strat}}\}$
end while

return $\left((x_i, w_i)_{i=1,\ldots,N}, \hat{Z} = \frac{1}{N}\sum_{i=1}^{N} w_i\right)$

caching") in single-particle settings, e.g. to enable conversational interfaces without re-evaluating the entire conversation history with each new message. But we have found that extending it to the multi-particle setting makes inference in language model probabilistic programs significantly cheaper, compared to previous approaches to integrating Transformer models into probabilistic programs [Lew et al., 2020, Dohan et al., 2022, Zhi-Xuan, 2022].

- **Without-replacement resampling.** Standard sequential Monte Carlo implementations maintain a fixed number of particles throughout their execution, and use randomized resampling strategies to clone some particles and cull others. Because of this, it is common for the same state $S_t = s_t$ to be represented multiple times within a particle collection. To maintain better particle diversity, we instead apply a resampling strategy that more closely resembles beam search, while maintaining the unbiasedness and posterior consistency properties we expect of SMC. At each step, any active particles (i.e., those not already terminated by `EOS`) are cloned K times, and each clone is independently extended using M_t. This larger collection of particles is then down-sampled back to size N, using a without-replacement down-sampling algorithm to ensure uniqueness of the N resampled particles. The down-sampling algorithm is close to that of Fearnhead and Clifford [2003], except that (1) we update the weights slightly differently to ensure they remain unbiased estimates of the partition function Z, and (2) we include new weight corrections to account for the fact that in our setting, some particles are *stopped* and thus not cloned during the expansion step.

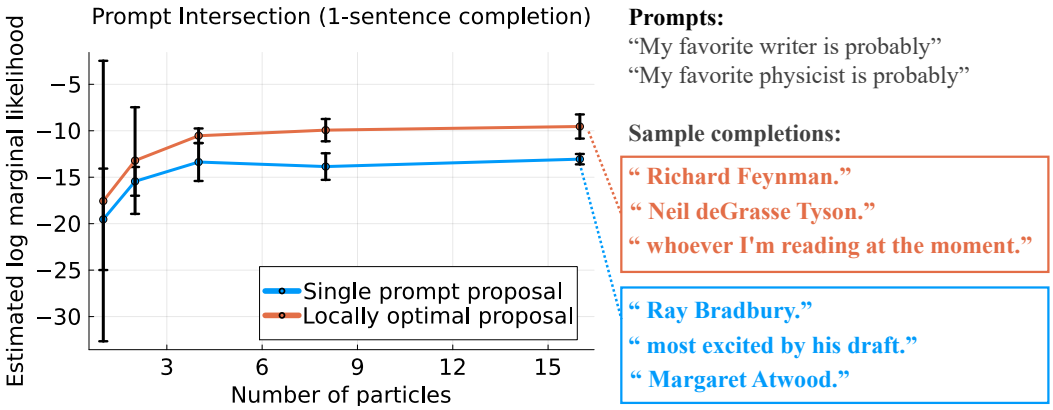

Figure 4: Results of SMC steering on the *prompt intersection* task from §2.2, modified to emit `EOS` after one sentence. **Left:** We plot mean values of $\log \hat{Z}$ across 10 runs of SMC steering, with varying numbers of particles N, fixed expansion factor $K = 3$, and the two Feynman-Kac models for prompt intersection given in §2.2. In the first model, the Markov kernel M_t proposes tokens according to only the first prompt (“My favorite writer is probably”), and the potential G_t conditions on agreement from the second prompt. In the second model, the Markov kernel M_t samples a locally optimal proposal distribution based on logits from both prompts, and G_t serves as an importance weight. **Right:** Higher $\mathbb{E}[\log \hat{Z}]$ corresponds to qualitatively better samples. Indeed, by Jensen's inequality, $\mathbb{E}[\log \hat{Z}]$ is a lower bound on $\log Z$, and the *gap* is itself an upper bound on the KL divergence between SMC steering's sampling distribution and the true posterior $\mathbb{P}$.

Accuracy. Under mild assumptions, SMC is *consistent* in the sense that as the number of particles grows, the marginal distribution over returned particles approaches the true posterior. It is also *unbiased* in that up to a normalizing constant, the expected value of the weighted average $\frac{1}{N}\sum w_i f(x_i)$ is the posterior expectation of f, for any measurable f. However, the variance of these estimates, and the accuracy of posterior sampling, depend on the Feynman-Kac model. Figure 4 illustrates this phenomenon in the *prompt intersection* task from §2.2. Recall that we formulated two Feynman-Kac models for this task, which targeted the same posterior $\mathbb{P}$ but made distinct choices of M_t and G_t. One way of understanding their differences is that they encode different *proposal distributions* for the task: the first model proposes according to a single prompt, whereas the second model proposes based on logits from all the prompts. As Figure 4 shows, the second outperforms the first.

A general design principle is that the proposal (i.e., the distribution from which M_t samples) should be as close to the posterior as possible. Nearly any heuristics for solving the generation task can be incorporated into the proposal, so long as the potentials G_t are appropriately modified to perform the proper importance weighting. One approach to developing better proposals for challenging generation tasks could be to propose tokens from an LLM with an auxiliary prompt that describes the task in natural language (where the prompt itself could be automatically generated via inference, as in Zhi-Xuan [2022] and Zhou et al. [2023]); then SMC steering would be responsible only for correcting mistakes made by the prompted proposal, rather than solving the task from scratch. Another approach could be to use “chain-of-thought” proposals, i.e., LLMs prompted to perform some reasoning before proposing tokens to solve the task [Wei et al., 2022]. Because the marginal probability of proposing a token would not be tractable for such a proposal, the potentials G_t would not be able to incorporate exact importance weights, and would instead need to approximate them unbiasedly. Future versions of `LLaMPPL` could incorporate recently introduced probabilistic programming techniques for automating this unbiased proposal density estimation [Lew et al., 2022, 2023].

4 Related Work and Discussion

Probabilistic programming with language models. To our knowledge, the idea of integrating language models as primitives into a probabilistic programming system was first proposed by Lew et al. [2020], who showed that in certain verbal reasoning tasks, the posteriors of such programs were better models of human behavior than unconstrained language models. More recently, Dohan

et al. [2022] proposed unifying various approaches to "chaining" LLMs by understanding them as graphical models or probabilistic programs with string-valued random variables. But in the "chain-of-thought"-style applications they explore, there are typically no unknown variables with non-trivial likelihood terms, so no inference algorithm is required—"forward" or "ancestral" sampling suffices.

Our examples, by contrast, induce non-trivial posteriors that require more powerful inference algorithms to sample. Zhi-Xuan [2022]'s `GenGPT3.jl` library integrates OpenAI's GPT-3 models into the `Gen.jl` probabilistic programming system [Cusumano-Towner et al., 2019], and includes examples of using `Gen.jl`'s posterior inference machinery to perform structured infilling (e.g., inferring which of a set of questions was likely to lead an observed answer, similar to automatic prompt engineering Zhou et al. [2023]) and constrained semantic parsing. However, the sequential Monte Carlo algorithms we describe here would be difficult to implement efficiently using `GenGPT3.jl`. One challenge is that the OpenAI API is stateless, and so "one-token-at-a-time" generation and conditioning would require prohibitively many calls (each with growing numbers of context tokens).

Steering language models with programs. Beurer-Kellner et al. [2022] recently coined the term *language model programming*, to refer to the use of specialized programming languages to guide the behavior of LLMs. Several such programming languages have since been introduced, including their SQL-inspired *language model query language* (LMQL), Microsoft and Lundberg [2023]'s Guidance language, and Normal Computing and Louf [2023]'s Outlines language. All three of these provide high-level DSLs that make chaining multiple calls to LLMs more ergonomic, and Outlines and LMQL also expose some features for generation subject to hard constraints. However, they do not support sampling the posterior *given* these constraints, only beam search (in the case of LMQL) and greedy decoding, using token masking to enforce the constraints. Furthermore, the constraint DSLs supported by these languages are limited, and cannot, for example, encode our prompt intersection task. That said, they support many features that would be useful to include in future versions of `LLaMPPL` (or higher-level DSLs built on it): a unified frontend to a variety of Transformer backends, automated computation of token masks for enforcing common constraints, and high-level syntax for chaining prompts together that does not require explicit token-by-token processing logic.

Controlled generation and probabilistic inference in language models. Many recent papers have proposed methods for more controlled text generation from language models, either through the lens of optimization [Kumar et al., 2021] or probabilistic inference [Kumar et al., 2022]. Approaches applied during fine-tuning include direct preference optimization [Rafailov et al., 2023], reinforcement learning from human feedback [Ouyang et al., 2022], and generation with distributional control [Khalifa et al., 2021], all of which can be viewed as forms of variational Bayesian inference due to their use of KL-divergence penalties [Korbak et al., 2022]. Finetuning methods have the benefit of avoiding increased runtime during decoding, but they typically cannot handle hard constraints, motivating the use of controlled generation at decoding time.

Among decoding-time approaches, many are focused on optimization, either through beam search [Meister et al., 2020] and heuristic search [Lu et al., 2021, Zhang et al., 2023b], or through gradient-based optimization in embedding space [Dathathri et al., 2019, Kumar et al., 2021]. Other approaches focus on sampling from a constrained or modified distribution [Zhang et al., 2023a], including naive rejection sampling [Poesia et al., 2022], but also more sophisticated Markov Chain Monte Carlo (MCMC) samplers [Miao et al., 2019, Hie et al., 2022] that make use of specialized proposal distributions [Zhang et al., 2020] or the gradients of continuous embeddings [Qin et al., 2022, Kumar et al., 2022]. However, a downside of MCMC methods is that they require potentially many iterations before producing a sample from the desired distribution, limiting their speed and usefulness. In contrast, SMC steering maintains the autoregressive nature of both regular decoding and beam search while still allowing constraints to be applied. As such, our method achieves the same overhead as beam search, while continuing to sample from the desired distribution instead of optimizing. This enables SMC steering to generate a diversity of constrained completions without the need for additional machinery [Vijayakumar et al., 2016].

Researchers have also proposed using the posterior distributions of probabilistic generative models defined in part using Transformers for tasks beyond constrained generation, e.g. semantic segmentation and household navigation, where the general world knowledge learned by the LLM is used to inform priors [Li et al., 2023]. Probabilistic programming tools like `LLaMPPL`, which support building models that use LLMs and performing efficient inference in them, could help make such approaches more accessible and scalable.

References

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *arXiv preprint arXiv:2212.06094*, 2022.

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 221–236, 2019.

Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. Plug and play language models: A simple approach to controlled text generation. *arXiv preprint arXiv:1912.02164*, 2019.

Pierre Del Moral. *Feynman-kac formulae*. Springer, 2004.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A Saurous, Jascha Sohl-Dickstein, et al. Language model cascades. *arXiv preprint arXiv:2207.10342*, 2022.

Chris Donahue, Mina Lee, and Percy Liang. Enabling language models to fill in the blanks. *arXiv preprint arXiv:2005.05339*, 2020.

Paul Fearnhead and Peter Clifford. On-line inference for hidden markov models via particle filters. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 65(4):887–899, 2003.

Brian Hie, Salvatore Candido, Zeming Lin, Ori Kabeli, Roshan Rao, Nikita Smetanin, Tom Sercu, and Alexander Rives. A high-level programming language for generative protein design. *bioRxiv*, pages 2022–12, 2022.

Muhammad Khalifa, Hady Elsahar, and Marc Dymetman. A distributional approach to controlled text generation. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=jWkw45-9AbL.

Tomasz Korbak, Ethan Perez, and Christopher Buckley. RL with KL penalties is better viewed as Bayesian inference. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 1083–1091, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.findings-emnlp.77.

Sachin Kumar, Eric Malmi, Aliaksei Severyn, and Yulia Tsvetkov. Controlled text generation as continuous optimization with multiple constraints. *Advances in Neural Information Processing Systems*, 34:14542–14554, 2021.

Sachin Kumar, Biswajit Paria, and Yulia Tsvetkov. Constrained sampling from language models via Langevin dynamics in embedding spaces. *arXiv preprint arXiv:2205.12558*, 2022.

Alexander K Lew, Michael Henry Tessler, Vikash K Mansinghka, and Joshua B Tenenbaum. Leveraging unstructured statistical knowledge in a probabilistic language of thought. In *Proceedings of the Annual Conference of the Cognitive Science Society*, 2020.

Alexander K Lew, Marco Cusumano-Towner, and Vikash K Mansinghka. Recursive monte carlo and variational inference with auxiliary variables. In *Uncertainty in Artificial Intelligence*, pages 1096–1106. PMLR, 2022.

Alexander K Lew, Matin Ghavamizadeh, Martin Rinard, and Vikash K Mansinghka. Probabilistic programming with stochastic probabilities. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2023.

Belinda Z Li, William Chen, Pratyusha Sharma, and Jacob Andreas. Lampp: Language models as probabilistic priors for perception and action. *arXiv e-prints*, pages arXiv–2302, 2023.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, et al. Neurologic A* esque decoding: Constrained text generation with lookahead heuristics. *arXiv preprint arXiv:2112.08726*, 2021.

Clara Meister, Tim Vieira, and Ryan Cotterell. If beam search is the answer, what was the question? *arXiv preprint arXiv:2010.02650*, 2020.

Ning Miao, Hao Zhou, Lili Mou, Rui Yan, and Lei Li. CGMH: Constrained sentence generation by metropolis-hastings sampling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6834–6842, 2019.

Microsoft and Scott Lundberg. Guidance: A guidance language for controlling large language models., 2023. URL https://github.com/microsoft/guidance.

Kenton Murray and David Chiang. Correcting length bias in neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 212–223, Brussels, Belgium, October 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-6322. URL https://aclanthology.org/W18-6322.

Normal Computing and Remi Louf. Outlines: Generative model programming, May 2023. URL https://github.com/normal-computing/outlines.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35: 27730–27744, 2022.

Damian Pascual, Beni Egressy, Florian Bolli, and Roger Wattenhofer. Directed beam search: Plug-and-play lexically constrained language generation. *arXiv preprint arXiv:2012.15416*, 2020.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.

Peng Qian and Roger Levy. Flexible generation from fragmentary linguistic input. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8176–8196, 2022.

Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. COLD decoding: Energy-based constrained text generation with Langevin dynamics. *arXiv preprint arXiv:2202.11705*, 2022.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.

Allen Roush, Sanjay Basu, Akshay Moorthy, and Dmitry Dubovoy. Most language models can be poets too: An ai writing assistant and constrained text generation studio. In *Proceedings of the Second Workshop on When Creative AI Meets Conversational AI*, pages 9–15, 2022.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*, 2016.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. Tractable control for autoregressive language generation. *arXiv preprint arXiv:2304.07438*, 2023a.

Maosen Zhang, Nan Jiang, Lei Li, and Yexiang Xue. Language generation via combinatorial constraint satisfaction: A tree search enhanced Monte-Carlo approach. *arXiv preprint arXiv:2011.12334*, 2020.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023b.

Tan Zhi-Xuan. GenGPT3.jl: GPT-3 as a generative function in Gen.jl, November 2022. URL https://github.com/probcomp/GenGPT3.jl.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=92gvk82DE-.