Introduction à la programmation probabiliste

Guillaume Baudart, Christine Tasson

Introduction à la programmation probabiliste

Guillaume Baudart et Christine Tasson

La programmation probabiliste est un paradigme qui a connu un essor important ces dernières années. Les langages probabilistes manipulent l'incertitude de manière explicite. Ils reposent sur la méthode bayésienne qui permet d'apprendre la distribution des paramètres d'un modèle à partir d'observations statistiques. De nombreux langages de programmation reposant sur ce paradigme ont été développés : WebPPL, Venture, Anglican, Stan, Gen, Pyro, Turing.jl,... Ces langages sont utilisés dans des domaines qui vont de la vision par ordinateur (génération d'images) et la robotique (planification), à la santé (épidémiologie) et les sciences sociales (sondages).

Ces notes présentent les concepts fondamentaux de la programmation probabiliste :

1. *Introduction à la modélisation bayésienne*
2. *Conception d'un langage probabiliste et d'un moteur d'inférence*
3. *Sémantiques formelles pour raisonner sur les programmes probabilistes*

Table des matières

1 Introduction

Le but d'un langage de programmation probabiliste est de décrire des variables aléatoires par des programmes. L'exécution d'un programme probabiliste construit la loi de la variable aléatoire qui lui est associée, afin de pouvoir la simuler, de calculer son espérance, sa variance ou sa fonction de répartition (de façon exacte ou approchée).

Un langage probabiliste enrichit un langage de programmation généraliste avec des constructions probabilistes : des distributions discrètes (`Bernoulli(p)`, `Binomial(n, p)`, `RandInt(n, m)`, ...), des distributions continues à densité (`Uniform(a, b)`, `Gaussian(m, s)`, ...), et des opérateurs probabilistes qui permettent de décrire des *modèles* :

— `sample` pour simuler un échantillon aléatoire dans une distribution,

— `assume`, `factor`, `observe` pour conditionner le modèle sur certaines hypothèses,

— `infer` pour calculer la loi de la variable aléatoire associée à un modèle.

Dans cette partie, nous donnons un sens plus précis à chacune de ces commandes à l'aide d'exemples de programmes probabilistes.

1.1 Programmes probabilistes et variables aléatoires

Le programme probabiliste ci-dessous décrit « *la somme des valeurs renvoyées par deux dés à six faces non pipés* ».

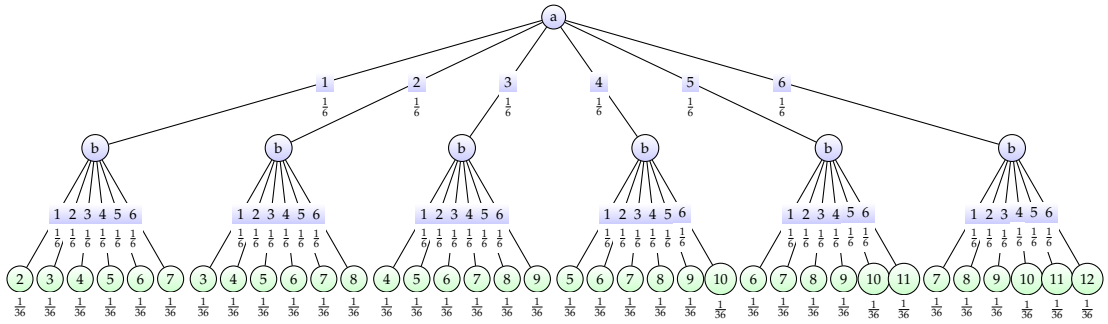
```
def sum_dice() → int:
    a = sample(RandInt(1, 6), name="a")
    b = sample(RandInt(1, 6), name="b")
    return a + b

with Enumeration():
    dist: Categorical = infer(sum_dice)
```

Il est constitué de deux parties. La première définit la fonction `sum_dice` qui simule la variable aléatoire. La commande [1] `sample(RandInt(1, 6))` permet de simuler les dés suivant une loi uniforme sur $\{1, \ldots, 6\}$.

La seconde partie définit `dist`, la loi de la variable aléatoire associée à `sum_dice`. C'est une distribution *catégorique* représentée par une liste de valeurs associées à leurs probabilités. La commande `with Enumeration(): ... infer(sum_dice)` calcule la loi associée à la fonction `sum_dice` en utilisant un *algorithme d'inférence* par énumération. Cet algorithme commence par lister toutes les *exécutions* possibles, c'est-à-dire les valeurs possibles prises par les dés. À chaque exécution, on associe une probabilité et une valeur de retour. Pour obtenir la loi, on associe à chaque valeur de retour la somme des probabilités des exécutions qui produisent cette valeur.

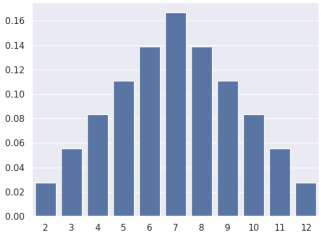
On peut représenter graphiquement l'algorithme d'inférence par énumération par l'arbre des exécutions possibles.

1. L'argument `name` de `sample` associe à l'échantillon un nom unique qui est utilisé pour l'inférence par énumération, nous pouvons l'ignorer dans cette introduction



Les branches des deux premiers niveaux de l'arbre sont étiquetées par les couples (v_i, w_i) où w_i (ici $\frac{1}{6}$) est la probabilité d'obtenir la valeur v_i (encadrée sur fond bleu) à l'aide de l'opérateur `sample`. Les feuilles (encerclées sur fond vert) contiennent les valeurs de retour de la variable aléatoire. Le *score* associé sous chaque feuille est obtenu en multipliant les probabilités étiquetant les branches qui mènent à cette feuille.

Pour construire la fonction de masse représentée par l'histogramme ci-contre, il suffit de sommer pour chaque valeur de retour le score associé aux feuilles étiquetées par cette valeur.



$$\mathbb{P}(\texttt{sum\_dice} = s) = \sum_{a=1}^{6} \sum_{b=1}^{6} \mathbb{1}_{\{a+b=s\}} w_a w_b$$

1.2 Loi conditionnelle

Le programme probabiliste ci-dessous décrit « *la somme des valeurs renvoyées par deux dés à six faces non pipés, sachant que les deux dés ont renvoyé des valeurs distinctes* » :

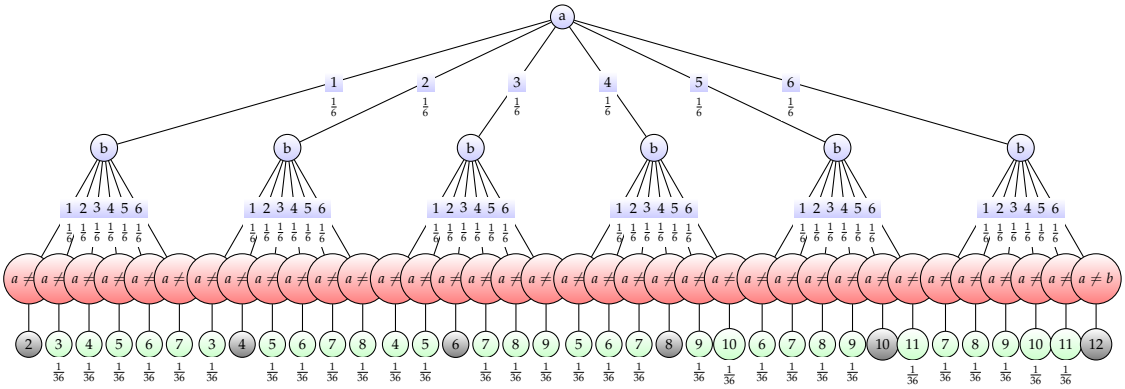
```
def hard_dice() → int:
    a = sample(RandInt(1, 6), name="a")
    b = sample(RandInt(1, 6), name="b")
    assume(a != b)
    return a + b

with Enumeration():
    dist: Categorical = infer(hard_dice)
```

La fonction `hard_dice` utilise la commande `assume(p)` qui permet de conditionner le modèle en ne simulant que des exécutions qui vérifient la propriété `p`. La loi de la variable aléatoire associée à la fonction `hard_dice` est la loi conditionnelle définie par :

$$\mathbb{P}(\texttt{hard\_dice} = s) = \mathbb{P}(\texttt{sum\_dice} = s \mid a \neq b) = \frac{\mathbb{P}((\texttt{sum\_dice} = s) \wedge a \neq b)}{\mathbb{P}(a \neq b)}$$

La commande `with Enumeration(): ... infer(hard_dice)` utilise à nouveau l'algorithme d'inférence par énumération pour calculer la loi conditionnelle de la variable

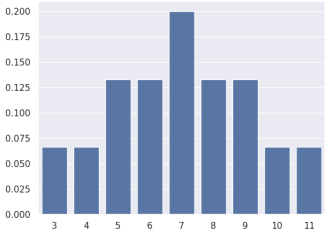
aléatoire associée à la fonction `hard_dice`. Cet algorithme d'inférence construit l'arbre des exécutions ci-dessous. Comparé à l'arbre de `sum_dice`, on a introduit un nouveau niveau correspondant à la commande `assume(a != b)`. Son rôle est de tester la propriété « *les dés ont renvoyé des valeurs distinctes* » qui étiquette les nouveaux nœuds.



Une fois construit l'arbre énumérant les exécutions, on peut calculer la loi de la variable aléatoire associée à `hard_dice`. Pour chaque valeur de retour, on somme les scores associés aux branches qui vérifient la propriété et on normalise la distribution en divisant par le score total :

$$\begin{aligned}\mathbb{P}(\texttt{hard\_dice} = s) &= \mathbb{P}(\texttt{sum\_dice} = s \mid a \neq b) \\ &= \frac{\displaystyle\sum_{a=1}^{6}\sum_{b=1}^{6} \mathbb{1}_{\{a+b=s\}} w_a w_b \mathbb{1}_{\{a\neq b\}}}{\displaystyle\sum_{a=1}^{6}\sum_{b=1}^{6} w_a w_b \mathbb{1}_{\{a\neq b\}}}\end{aligned}$$

On obtient une distribution catégorique similaire à celle de `sum_dice` mais où les valeurs qui correspondent à un double ne sont pas prises en compte (leur score est pénalisé à 0). L'histogramme de la fonction de masse de la variable aléatoire associée à la fonction `hard_dice` est affiché ci-contre.



La méthode d'inférence par énumération fonctionne avec des programmes probabilistes qui ne font intervenir que des lois discrètes à support fini. Dès que la combinatoire du modèle augmente, cette méthode ne peut plus être utilisée en pratique.

1.3 Loi des grands nombres et simulation de Monte-Carlo

Le programme probabiliste ci-dessous décrit une « *loi uniforme sur le disque centré en* 0 *et de rayon* 1 ».

```
def disk() → Tuple[float, float]:
    x = sample(Uniform(-1, 1))
    y = sample(Uniform(-1, 1))
    assume (x**2 + y**2 < 1)
    return (x, y)

with RejectionSampling(num_samples=1000):
    dist: Empirical = infer(disk)
```

Les commandes `sample` simulent une loi uniforme sur le carré $[-1,1]^2$ qui est conditionnée par l'opérateur `assume` sur les points dont le carré de la distance à l'origine est inférieur à 1. Dans la seconde partie du programme, la commande `with RejectionSampling ... infer(disk)` *approche* la loi de la variable aléatoire associée au programme `dist` par une distribution *empirique*, c'est-à-dire une distribution discrète à support fini dont toutes les valeurs sont équiprobables.

Les exécutions qui produisent des simulations de la loi uniforme sur le carré $[-1,1]^2$ ne peuvent pas être énumérées (car elles sont non-dénombrables). On utilise donc une *simulation de Monte-Carlo* : on construit un *n-échantillon* $(X_1,\dots,X_n)$, c'est-à-dire un tableau de n simulations indépendantes de la loi X. La *loi des grands nombres* assure alors que si $(X_i)_{i\in\mathbb{N}}$ est une famille de variables aléatoires *i.i.d.* de loi X et si f est une fonction telle que $f(X)$ admet une espérance, alors la moyenne empirique $\frac{1}{n}\sum_{i\leq n} f(X_i)$ converge presque sûrement vers l'espérance de $f(X)$.

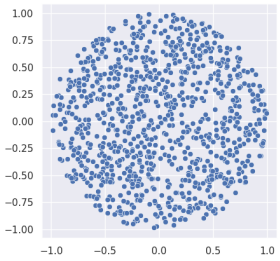
Par exemple,

— l'espérance de X, $\mathbb{E}(X)$ est approchée presque sûrement (*p.s.*) par $\frac{1}{n}\sum_{i\leq n} X_i$,

— la fonction de répartition $\mathbb{P}(X\leq x)$ est approchée *p.s.* par $\frac{1}{n}\sum_{i\leq n} \mathbb{1}_{\{X_i\leq x\}}$,

— la probabilité $\mathbb{P}(X\in U)$ est approchée *p.s.* par $\frac{1}{n}\sum_{i\leq n} \mathbb{1}_{\{X_i\in U\}}$.

Pour représenter la loi associée à la fonction `disk`, la commande `with RejectionSampling ... infer(disk)` crée un tableau de 1000 échantillons (figurés ci-contre). La taille du tableau est déterminée par le programmeur à l'aide du paramètre `num_samples=1000`. Chacun des échantillons est calculé avec un *algorithme de rejet* : *tant que l'on n'a pas simulé un point du carré qui est à l'intérieur du disque, on exécute la fonction* `disk`. L'algorithme de rejet termine *p.s.* [2].



2. En notant p la probabilité d'un rejet, alors la probabilité d'obtenir une position dans le disque en $n+1$ itérations de la boucle est $p^n(1-p)$. En sommant ces probabilités, on obtient une probabilité de retourner un succès en un temps fini égale à 1 (même si ce temps peut être très long !).

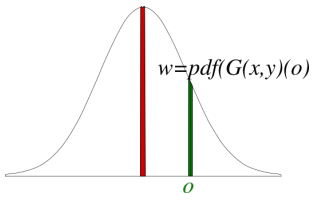
1.4 Conditionnement sur une observation

Le programme ci-dessous décrit « *la position d'un objet a priori réparti uniformément dans un carré, étant donnée sa distance à l'origine* », calculée par exemple à partir de la mesure bruitée d'un radar.

```
def position(o: float) → Tuple[float, float]:
    x = sample(Uniform(-1, 1))
    y = sample(Uniform(-1, 1))
    d2 = x*x + y*y
    observe(Gaussian(d2, 0.1), o)
    return (x, y)

with ImportanceSampling(num_particles=10000):
    dist: Categorical = infer(position, 0.5)
```

L'opérateur `sample` permet de simuler une variable aléatoire Z de loi uniforme sur le carré $[-1,1]^2$. L'opérateur `observe` permet de modéliser l'observation `o` suivant une variable aléatoire $G(x,y)$ de loi gaussienne centrée sur le carré de la distance à zéro du point $Z=(x,y)$ et de variance 0.1. Cette loi gaussienne modélise le bruit sur mesure par le radar de la distance. La fonction `position` modélise la loi de Z sachant $G(x,y)=o$.

La figure ci-contre présente la fonction de densité de $G(x,y)$. Sa moyenne est indiquée par la barre rouge. L'observation o est indiquée par la barre verte. On peut faire l'hypothèse que l'observation o a été simulée en suivant la loi $G(x,y)$ en utilisant le score $\texttt{pdf}(G(x,y))(o)$, c'est-à-dire la fonction de densité de $G(x,y)$ en o. Plus l'observation correspond à une forte densité, plus le score est important. L'opérateur `observe` est un raccourci syntaxique qui permet d'écrire directement $\texttt{observe}(G(x,y),o)$ plutôt que $\texttt{factor}(\texttt{pdf}(G(x,y))(o))$ pour modifier le score associé à chaque valeur simulée (x,y) et conditionner Z.



La commande `with ImportanceSampling ... infer(position, 0.5)` permet d'approcher la loi associée à la fonction `position` étant donnée l'observation `o=0.5` en utilisant l'algorithme d'*échantillonage préférentiel*. Cet algorithme produit une distribution catégorique. Chaque valeur est obtenue en exécutant la fonction `position`. Les opérateurs `sample` permettent de simuler la position (x,y) associée au score 1 (car la loi est uniforme). La commande `observe(Gaussian(d2, 0.1), o)` multiplie ce score par $\texttt{pdf}(G(x,y))(o)=\mathcal{N}(d2,0.1)(o)$ qui mesure la qualité des simulations par rapport au modèle.

1.5 Modèles hiérarchiques et formule de Bayes

Les modèles hiérarchiques sont des modèles qui simulent des lois dont les paramètres sont eux-mêmes des variables aléatoires.

Par exemple, étant donnée une pièce équilibrée et un entier s, on étudie la question : « *combien de fois doit on lancer la pièce pour obtenir le nombre de succès s ?* ». Plus précisément, étant donnée une variable aléatoire N de loi uniforme sur les entiers entre 10 et 20, on lance une pièce équilibrée N fois. On note $B(N, 0.5)$ le nombre de succès. La loi de $B(N, 0.5)$ sachant $\{N = n\}$ est une loi binomiale de paramètre n et de biais 0.5. Le programme ci-dessous décrit la loi conditionnelle de $N = n$ sachant $B(N, 0.5) = s$.

```
def success(s:int) → int:
    n = sample(RandInt(10, 20))
    observe(Binomial(n, 0.5), s)
    return n
```

Pour construire la loi de la variable aléatoire N conditionnée sur le nombre de succès, l'opérateur `sample` simule N, on parle de *loi a priori* (*prior distribution*). Chaque valeur simulée n est associée à un score correspondant à la probabilité $\mathbb{P}(N = n)$. L'opérateur `observe` multiplie cette probabilité en utilisant la *fonction de vraisemblance* (*likelihood*) :

$$\mathbb{P}(B(N, 0.5) = s \mid N = n) = \texttt{Binomial(n, 0.5)}(s).$$

On applique la formule de Bayes pour calculer la loi associée à la fonction `success` :

$$\begin{aligned}\mathbb{P}(N = n \mid B(N, 0.5) = s) &= \frac{\mathbb{P}(B(N, 0.5) = s \mid N = n)\,\mathbb{P}(N = n)}{\mathbb{P}(B(N, 0.5) = s)} \\ &= \frac{\mathbb{P}(B(N, 0.5) = s \mid N = n)\,\mathbb{P}(N = n)}{\sum_k \mathbb{P}(B(N, 0.5) = s \mid N = k)\,\mathbb{P}(N = k)}\end{aligned}$$

Par la formule de Bayes, on obtient un score $\mathbb{P}(B(N, 0.5) = s \mid N = n)\mathbb{P}(N = n)$ proportionnel à la probabilité recherchée $\mathbb{P}(N = n \mid B(N, 0.5) = s)$. Pour finir, il faut calculer la somme au dénominateur de la formule de Bayes pour normaliser la distribution et obtenir une loi de probabilité, que l'on appelle *loi a posteriori* (*posterior distribution*).

Le programme `success` est un modèle hiérarchique dans lequel le paramètre N est discret et fini. On peut donc énumérer toutes les exécutions et calculer explicitement cette somme.

La formule de Bayes peut aussi être utilisée dans le cas d'un paramètre continu.

On s'intéresse par exemple à une pièce biaisée et on recherche la loi du « *biais de la pièce étant donnée l'observation d'une série de tirages indépendants* ». Plus précisément, étant donnée une variable aléatoire P de loi uniforme sur le segment $[0, 1]$, on lance plusieurs fois une pièce de biais P. Les résultats des lancers sont notés (O_i). Ce sont des variables aléatoires *i.i.d.*. Soit `obs` la liste des valeurs observées o_i. On note $Obs = O_1 \wedge \cdots \wedge O_n$ la loi jointe et $Obs =$ `obs` l' intersection des évènements $O_i = o_i$. Le programme ci-dessous décrit la loi conditionnelle de P sachant $Obs =$ `obs`.

```
def coin(obs: list[int]) → float:
    p = sample(Uniform(0, 1))
```

```
    for o in obs:
        observe(Bernoulli(p), o)
    return p

with inference.ImportanceSampling(num_particles=10000):
    dist: Categorical = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
```

On applique la formule de Bayes formulée avec les fonctions de densité $\texttt{pdf}(P)$, $\texttt{pdf}(Obs)$ et de la densité de la loi conditionnelle $\texttt{pdf}(P \mid Obs)$:

$$\begin{aligned}\texttt{pdf}(P \mid Obs)(p \mid \texttt{obs}) &= \frac{\texttt{pdf}(Obs \mid P)(\texttt{obs} \mid p)\ \texttt{pdf}(P)(p)}{\texttt{pdf}(Obs)(\texttt{obs})} \\ &= \frac{\texttt{pdf}(Obs \mid P)(\texttt{obs} \mid p)\ \texttt{pdf}(P)(p)}{\int \texttt{pdf}(Obs \mid P)(\texttt{obs} \mid p)\ \texttt{pdf}(P)(p)dp}\end{aligned}$$

À nouveau, pour construire la loi du paramètre P conditionnée aux observations des lancers $Obs = \texttt{obs}$ on simule des exécutions du modèle. L'opérateur `sample` tire une valeur aléatoire pour le paramètre $p \in [0,1]$. Les commandes `observe` pondèrent la valeur de retour de l'exécution en utilisant la densité de la loi `Bernoulli(P)` pour chacune des observations $o = \texttt{pile}$ ou $o = \texttt{face}$ de obs. Le score associé à l'exécution est alors de :

$$\begin{aligned}\texttt{pdf}(Obs \mid P)(\texttt{obs} \mid p) &= \prod_{o \in \texttt{obs}} \texttt{pdf}(\texttt{Bernoulli}(P) \mid P)(o \mid p) \\ &= p^{\#\texttt{pile}}(1-p)^{\#\texttt{face}}\end{aligned}$$

L'intégrale $\int p^{\#\texttt{pile}}(1-p)^{\#\texttt{face}}dp$ au dénominateur de la formule de Bayes peut être calculée directement. On reconnaît alors la fonction de densité de la distribution Bêta :

$$\texttt{Beta}(\#\texttt{pile}+1, \#\texttt{face}+1) = \frac{p^{\#\texttt{pile}}(1-p)^{\#\texttt{face}}}{\int p^{\#\texttt{pile}}(1-p)^{\#\texttt{face}}dp}$$

En pratique, on a rarement la chance de savoir calculer l'intégrale. Les algorithmes d'inférence approchés permettent d'en obtenir une approximation.

Un modèle hiérarchique introduit par Laplace. Dans son ouvrage *Théorie analytique des probabilités* [Laplace, 1812, p. 379], Laplace étudie les nombres de naissances à Paris et à Londres au XVIII$^{\text{e}}$ siècle. Il montre à l'aide de calculs analytiques que « *la probabilité que la proportion de garçons baptisés soit plus grande à Londres qu'à Paris* » est de $1 - \frac{1}{328269}$. Pour cela, Laplace considère que le nombre de baptêmes de garçons est un paramètre probabiliste suivant *a priori* une loi uniforme sur $[0,1]$ dont on peut calculer la loi sachant la donnée du nombre de baptêmes naissances de chaque sexe observé à Londres et à Paris sur des grandes périodes. Ce sont les prémisses de l'inférence bayésienne.

L'idée principale, au cœur de l'inférence bayésienne est de regarder les paramètres inconnus comme des variables aléatoires dont il faut inférer la loi conditionnée aux données observées. Laplace utilise l'analyse pour calculer des intégrales comme celle

au dénominateur de la formule de Bayes et obtenir un résultat exact. Parfois, ce calcul n'est pas possible directement et la programmation probabiliste permet d'approcher la loi conditionnelle par différentes méthodes d'inférence.

```
def laplace(f1: int, g1: int, f2: int, g2: int) → float:
    p = sample(Uniform(0, 1), name="p")
    q = sample(Uniform(0, 1), name="q")
    observe(Binomial(f1 + g1, p), g1, name="f1")
    observe(Binomial(f2 + g2, q), g2, name="f2")
    return q > p

fp, gp = 377555, 393386 # Paris 1745 – 1784
fl, gl = 698958, 737629 # Londres 1664 – 1758
with ImportanceSampling(num_particles=100000):
    dist: Categorical = infer(laplace, fp, gp, fl, gl)
```

Le programme `laplace` modélise cette expérience historique à l'aide d'un modèle hiérarchique. Il permet d'approcher la probabilité que « *la proportion de garçons est presque sûrement plus grande à Londres qu'à Paris* » et de répondre à la question de Laplace.

En résumé. Les exemples précédents montrent que la programmation probabiliste permet de décrire des modèles probabilistes par des programmes qui peuvent faire intervenir toutes les constructions classiques des langages de programmation comme les boucles ou les conditionnelles. Le moteur d'inférence permet ensuite de calculer ou d'approcher automatiquement la distribution conditionnelle décrite par un modèle.

La conception d'un langage probabiliste et du moteur d'inférence est présentée Section 2. La sémantique permet d'étudier les propriétés de correction du langage et de raisonner sur les programmes, elle est présentée Section 3.

2 Implémentation

En général, la distribution *a posteriori* décrite par un modèle probabiliste n'est pas calculable. Ce problème est même NP-difficile pour le cas limité des réseaux bayésiens où toutes les variables aléatoires sont booléennes [Cooper, 1990]. Les langages probabilistes se concentrent donc sur des sous-classes de modèles pour lesquels ils peuvent atteindre des performances satisfaisantes : modèles discrets (Dice [Holtzen *et al.*, 2020]), modèles continus et différentiables (Stan [Carpenter *et al.*, 2017]), modèles calculables (Psi [Gehr *et al.*, 2016]). Plus le modèle est complexe, plus il nécessite un algorithme d'inférence avancé. Certains langages proposent donc une collection de méthodes d'inférence : WebPPL [Goodman et Stuhlmüller, 2014], Pyro [Bingham *et al.*, 2019], PyMC [Salvatier *et al.*, 2016], Gen.jl [Cusumano-Towner *et al.*, 2019]. C'est à l'utilisateur de choisir une méthode adaptée en fonction du modèle.

Dans cette section nous décrivons l'implémentation de μ-PPL, un prototype de langage probabiliste en Python avec plusieurs méthodes d'inférence approchées classiques. Nous illustrons ainsi comment utiliser les algorithmes d'inférence pour estimer

la distribution décrite par un programme probabiliste. Le code complet ainsi que les exemples sont disponibles en ligne : `https://github.com/gbdrt/mu-ppl`.

2.1 Infrastructure

Pour un modèle donné, l'évaluation des opérateurs probabilistes dépend de la méthode d'inférence choisie. Une méthode d'inférence correspond donc à un interprète capable d'évaluer un modèle probabiliste.

En μ-PPL, chaque méthode d'inférence est une classe qui hérite d'une classe spéciale `Handler` et qui définit une méthode pour chaque opérateur probabiliste : `sample`, `assume`, `factor`, `observe` et `infer`. Les instances de `Handler` sont des *context managers* qui permettent d'interpréter différemment les opérateurs probabilistes en fonction du contexte d'appel du modèle [Obermeyer et Bingham, 2020]. En Python, un contexte est introduit avec la commande `with`. Par exemple, le code suivant teste d'abord l'échantillonnage préférentiel puis la méthode du rejet sur le modèle `coin` présenté en page 8.

```
with ImportanceSampling(num_particles=1000):
    dist = infer(coin, [0, 0, 1, 1, 0])

with RejectionSampling(num_samples=1000):
    dist = infer(coin, [0, 0, 1, 1, 0])
```

2.2 Échantillonnage préférentiel et méthode du rejet

Pour implémenter des méthodes d'inférence approchées, une technique classique consiste à interpréter le modèle comme un *simulateur pondéré* : chaque exécution renvoie un échantillon v_i associé à un score W_i qui mesure la qualité d'une exécution par rapport au modèle.

Ce simulateur sert de base à de nombreux algorithmes d'inférence. Dans cette section nous présentons les plus élémentaires : l'*échantillonnage préférentiel* et la *méthode du rejet*. Les méthodes de Monte-Carlo par chaîne de Markov (MCMC) sont discutées dans la Section 2.3.

Échantillonnage préférentiel. La commande `sample(dist)` renvoie une valeur tirée aléatoirement dans la distribution *a priori* `dist`. Le score est mis à jour par les opérateurs de conditionnement. La commande `assume(p)` multiplie le score par 0 si la condition est fausse (cette exécution est impossible pour le modèle). La commande `factor(w)` multiplie le score par un facteur arbitraire `w` pour favoriser (ou défavoriser) une exécution. Enfin, `observe(dist, v)` multiplie le score par la *vraisemblance* de la valeur `v` pour la distribution `dist`, c'est-à-dire la valeur de la densité de `dist` au point `v` (on fait ainsi l'hypothèse que la valeur `v` a été échantillonnée dans la distribution `dist`). La commande `infer(model, *args)` lance n exécutions indépendantes du modèle, appelées *particules* [3].

3. `*args` indique un nombre arbitraire d'arguments en Python. Ici ce sont les arguments du modèle.

```
class ImportanceSampling(Handler):
    def __init__(self, num_particles):
        self.num_particles = num_particles
        self.score = 0 # current score

    def sample(self, dist):
        return dist.sample() # draw a sample

    def assume(self, p):
        if not p: self.score += -inf # zero probability

    def factor(self, weight):
        self.score += weight # update the score

    def observe(self, dist, v):
        self.score += dist.log_prob(v) # update the score

    def infer(self, model, *args):
        samples = []
        for i in range(self.num_particles):
            self.score = 0 # reset the score
            value = model(*args) # run the model
            samples.append(value, self.score) # log the result
        return Categorical(samples)
```

FIGURE 1 – Échantillonnage préférentiel

On peut ensuite calculer une probabilité p_i pour chaque v_i en normalisant les scores W_i :

$$p_i = \frac{W_i}{\sum_{1 \leq i \leq n} W_i}$$

On obtient ainsi une distribution *catégorique* qui approche la distribution décrite par le modèle, c'est-à-dire un ensemble de valeurs chacune associée à une probabilité.

En μ-PPL, on utilise les attributs du `Handler` pour stocker toutes les informations nécessaires pour l'inférence : ici, le score de la particule en cours d'exécution. Le code est présenté en Figure 1. Pour des raisons de stabilité numérique, on calcule les scores en échelle logarithmique.

Exemple. *Pour illustrer cette méthode d'inférence, considérons le modèle suivant qui cherche à estimer une position (x, y) à partir d'une série d'observations bruitées. On fait l'hypothèse que chaque observation est tirée dans une distribution gaussienne centrée sur (x, y).*

```
def gauss(obs: List[Tuple[float, float]]) → Tuple[float, float]:
    x = sample(Gaussian(0, 10))
    y = sample(Gaussian(0, 10))
    for (xi, yi) in obs:
        observe(Gaussian(x, 1), xi)
        observe(Gaussian(y, 1), yi)
    return x, y
```

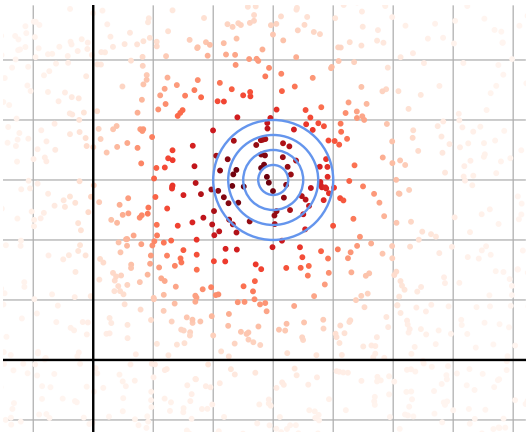


FIGURE 2 – Estimation d'une position à partir d'observations bruitées avec `ImportanceSampling` et 10 000 particules. Plus la couleur est foncée, plus le score est important.

Le résultat de l'inférence est présenté en Figure 2. La distribution a priori *donne très peu d'information : les positions sont simulées dans une gaussienne avec une très grande variance (une distribution quasiment uniforme sur le domaine affiché). La couleur représente le score de chaque échantillon. On constate que le score des échantillons tend rapidement vers* 0 *quand on s'éloigne du voisinage de la distribution attendue (représentée par les lignes de niveau bleues).*

Méthode du Rejet. Cette méthode permet de générer des échantillons de la distribution décrite par le modèle. C'est une généralisation de la méthode du rejet présentée en page 4 : on tire un ensemble d'échantillons aléatoires et on ne garde que ceux qui respectent le modèle. L'interprétation des commandes `sample`, `assume` et `observe` est la même que pour l'échantillonnage préférentiel. Ces méthodes sont directement héritées de la classe `ImportanceSampling`. La commande `infer(model, *args)` exécute le modèle autant de fois que nécessaire pour générer n échantillons. On note $W_{\max}$ le score de la meilleure exécution possible. À chaque étape :

1. On exécute le modèle pour obtenir un candidat v et un score W.
2. On tire $u \sim U(0,1)$ (loi uniforme sur $[0,1]$).
3. On accepte l'échantillon si $u \leq W/W_{\max}$, sinon on retourne à l'étape 1. La probabilité d'accepter le candidat est donc $W/W_{\max}$.

On renvoie ensuite une distribution *empirique*, c'est-à-dire un ensemble d'échantillons. Le code est présenté en Figure 3.

```python
class RejectionSampling(ImportanceSampling):
    def __init__(self, num_samples, max_score):
        self.num_samples = num_samples
        self.max_score = max_score # max score for the model
        self.score = 0 # current score

    # sample, assume, factor, observe inherited from ImportanceSampling

    def infer(self, model, *args):
        samples = []
        while (len(samples) < self.num_samples):
            self.score = 0 # reset the score
            value = model(*args) # generate sample
            u = np.random.random()
            if u <= np.exp(self.score - self.max_score):
                samples.append(value) # accept
        return Empirical(samples)
```

FIGURE 3 – Méthode du rejet

Il est parfois nécessaire d'exécuter le modèle bien plus que n fois pour obtenir n échantillons. Pour certains modèles conditionnés sur des évènements rares, comme dans l'exemple gauss page 11, la plupart des échantillons ont un score extrêmement faible et la probabilité de rejet est très grande. En pratique, l'inférence converge donc très lentement. Par ailleurs, il parfois difficile d'estimer le score maximal.

Ces deux méthodes classiques montrent rapidement leurs limites dès que le modèle fait intervenir plusieurs variables aléatoires. Il est en effet de plus en plus difficile d'échantillonner aléatoirement un espace à mesure que la dimension augmente : un problème connu sous le nom de *fléau de la dimension*.

2.3 MCMC et Metropolis-Hastings

Les méthodes de Monte-Carlo par chaînes de Markov (MCMC) on été introduites pour approcher efficacement des modèles de grande dimension. Comme la méthode du rejet, ces méthodes permettent de générer des échantillons d'une distribution qu'on ne sait pas calculer. Ces méthodes sont aujourd'hui au cœur des moteurs d'inférence des langages probabilistes les plus populaires comme Stan ou PyMC.

L'algorithme de Metropolis-Hastings [Hastings, 1970] construit une chaîne de Markov sur les exécutions possibles du modèle qui converge vers la distribution recherchée. Nous adaptons ici la présentation de [van de Meent *et al.*, 2018, Section 4.2] à μ-PPL.

On appelle X l'ensemble des variables aléatoires du modèle et $P(X)$ la distribution *a priori* associée (définie par les opérateurs `sample`). Un ensemble d'échantillons aléatoires pour X caractérise exactement une exécution du modèle. L'algorithme est le suivant. On tire aléatoirement une première exécution X_0 pour initialiser la chaîne. On obtient ainsi une première valeur v_0 et un score W_0. Puis, à chaque étape :

1. On tire aléatoirement une nouvelle exécution suivant une *probabilité de transition* Q : $X' \sim Q(X' \mid X_i)$ pour obtenir un candidat v' et un score W'.
2. On calcule le taux d'acceptation : $\alpha = \dfrac{P(X')\ W'\ Q(X_i \mid X')}{P(X_i)\ W_i\ Q(X' \mid X_i)}$.
3. On tire aléatoirement $u \sim U(0, 1)$.
4. Si $u \leq \alpha$, on *accepte* le candidat : $v_{i+1} = v'$, $X_{i+1} = X'$ et $W_{i+1} = W'$, sinon, on *rejette* le candidat et on garde le résultat précédent : $v_{i+1} = v_i$, $X_{i+1} = X_i$ et $W_{i+1} = W_i$. La probabilité d'accepter le candidat est donc $\min(1, \alpha)$.

Propositions indépendantes. Le taux d'acceptation α dépend de la probabilité de transition $Q(X' \mid X_i)$ choisie. Une solution naïve consiste à tirer aléatoirement un nouvel échantillon pour chaque variable aléatoire dans la distribution *a priori*. Le candidat X' ne dépend alors plus de l'exécution précédente X_i. On a alors $Q(X' \mid X_i) = P(X')$, $Q(X_i \mid X') = P(X_i)$ et :

$$\alpha = \frac{P(X')\ W'\ P(X_i)}{P(X_i)\ W_i\ P(X')} = \frac{W'}{W_i}$$

Le taux d'acceptation correspond donc au rapport entre le score du candidat et le score précédent. Si le score du candidat est meilleur que le score précédent, on l'accepte. Si le score du candidat est très inférieur au score précédent, le candidat à beaucoup de chances d'être rejeté.

L'interprétation des commandes `sample`, `assume`, `observe` est la même que pour l'échantillonnage préférentiel : on tire des valeurs aléatoires dans les distributions *a priori*, et on met à jour le score. À chaque étape, la commande `infer(model, *args)` exécute le modèle pour obtenir un candidat qui peut être accepté ou rejeté. Le comportement est donc très proche de `RejectionSampling`, mais plutôt que de boucler si le candidat est rejeté, on duplique l'échantillon précédent. Le code est présenté en Figure 4.

Exemple. *Le résultat de l'inférence sur l'exemple* `gauss` *page 11 est présenté en Figure 5. La chaîne converge bien vers la distribution attendue. On constate cependant qu'après 7000 itérations, on n'observe que quelques dizaines de points différents. À chaque itération, la chaîne tire des valeurs aléatoires pour* `x` *et* `y` *dans* `Gaussian(0, 10)`*. Lorsque la chaîne a convergé, il y a peu de chance de tomber sur des valeurs qui permettent d'augmenter le score. La probabilité de rejet est donc très importante et les meilleurs échantillons sont très souvent dupliqués.*

Probabilité de transition. La stratégie d'exploration de la solution précédente est très naïve. À chaque étape, le candidat est complètement indépendant de l'exécution précédente. Une alternative populaire pour les langages de programmation probabilistes consiste à réutiliser autant que possible les échantillons tirés lors de l'exécution précédente pour générer un candidat [Wingate *et al.*, 2011]. Plutôt que de tirer aléatoirement une nouvelle valeur pour toutes les variables aléatoires, on ne ré-échantillonne qu'une seule variable x_{regen}.

```python
class SimpleMetropolisHastings(ImportanceSampling):
    def __init__(self, num_samples):
        self.num_samples = num_samples
        self.score = 0 # current score

    # sample, assume, factor, observe inherited from ImportanceSampling

    def infer(self, model, *args):
        samples = []
        new_value = model(*args) # generate first sample
        for _ in range(self.num_samples):
            p_score = self.score # store state
            p_value = new_value
            self.score = 0 # reset the score
            new_value = model(*args) # generate a candidate
            alpha = np.exp(self.score - p_score)
            u = np.random.random()
            if not (u <= alpha): # reject
                self.score = p_score # rollback
                new_value = p_value
            samples.append(new_value) # store sample
        return Empirical(samples)
```

FIGURE 4 – Metropolis-Hastings avec propositions indépendantes

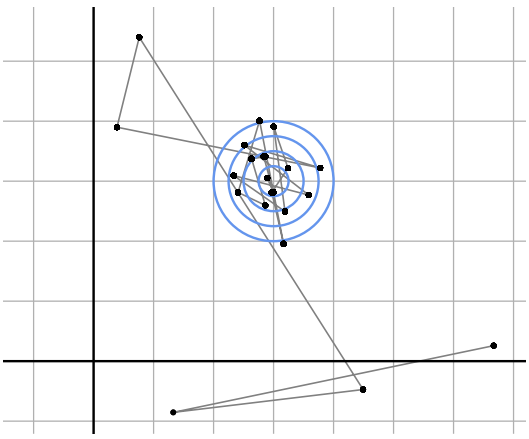


FIGURE 5 – Estimation d'une position à partir d'observations bruitées avec 7000 itérations de `SimpleMetropolis`.

Pour mesurer la qualité des échantillons réutilisés, on associe la densité $w(x)$ de l'échantillon à chaque variable $x \in X$ (comme pour `observe`). Le *cache* est l'ensemble des variables réutilisées entre X et X' : $C = (X' \cap X - \{x_{\text{regen}}\})$. On obtient alors :

$$P(X) = \prod_{x \in X} w(x) \qquad \textit{distribution a priori}$$

$$Q(X' \mid X) = \frac{1}{|X|} \prod_{x \in (X' - C)} w'(x) \qquad \textit{choix de } X' \textit{ à partir de } X$$

Ici, X et X' jouent des rôles symétriques. La probabilité d'obtenir X' à partir de X correspond au choix de x_{regen} (un choix parmi $|X|$) et au score de toutes les variables ré-échantillonnées. On peut maintenant calculer le taux d'acceptation.

$$\begin{aligned}
\alpha &= \frac{P(X')\, W'\, Q(X_i \mid X')}{P(X_i)\, W_i\, Q(X' \mid X_i)} \\
&= \frac{\prod_{x \in X'} w'(x)}{\prod_{x \in X_i} w(x)} \frac{W'}{W_i} \frac{|X_i| \prod_{x \in (X_i - C)} w(x)}{|X'| \prod_{x \in (X' - C)} w'(x)} \\
&= \frac{|X_i|}{|X'|} \frac{W'}{W_i} \frac{\prod_{x \in C} w'(x)}{\prod_{x \in C} w(x)}
\end{aligned}$$

Par rapport à la solution précédente, tout se passe comme si les variables réutilisées étaient traitées comme des observations supplémentaires.

En pratique, pour utiliser l'algorithme de Metropolis-Hastings, on associe un nom unique à chaque variable. Ces noms peuvent être ajoutés automatiquement par un compilateur (comme en WebPPL [Wingate *et al.*, 2011]), ou indiqués par l'utilisateur (comme en Pyro et PyMC, et μ-PPL). Par exemple, il faut réécrire l'exemple `gauss` de la manière suivante :

```
def gauss(obs: List[Tuple[float, float]]) → Tuple[float, float]:
    x = sample(Gaussian(0, 10), name="x")
    y = sample(Gaussian(0, 10), name="y")
    for (xo, yo) in enumerate(obs):
        observe(Gaussian(x, 1), xo)
        observe(Gaussian(y, 1), yo)
    return x, y
```

Grâce à ces noms uniques, l'algorithme peut maintenir trois tables : `cache` contient les échantillons réutilisées d'une exécution à l'autre, `x_samples` contient les échantillons de l'exécution courante et `x_scores` contient les scores associés. La commande `sample` réutilise si possible l'échantillon stocké dans le cache et enregistre le score dans `x_scores`. L'interprétation des commandes `assume`, `factor` et `observe` est à nouveau la même que pour l'échantillonnage préférentiel : on met à jour le score. À chaque itération, la commande `infer(model, *args)` :

1. sauvegarde le résultat de l'exécution précédente,
2. choisit une variable `regen` à ré-échantillonner,
3. exécute le modèle pour obtenir un nouveau candidat en utilisant les échantillons de l'exécution précédente comme cache,
4. accepte ou rejette le candidat.

Pour calculer le taux d'acceptation, la fonction auxiliaire `mh` utilise le score `p_score` et la table `p_x_scores` de l'exécution précédente, et le score `self.score` et la table `self.x_scores` du candidat. Le code est présenté en Figure 6.

Exemple. *Le résultat de l'inférence sur l'exemple* `gauss` *est présenté en Figure 7. La trajectoire « en escalier » est caractéristique de la stratégie d'exploration : chaque itération ne change qu'une seule variable,* `x` *ou* `y`*. Après seulement 1000 itérations on obtient déjà beaucoup plus de points différents que dans la Figure 5. La chaîne accepte plus facilement les candidats, l'algorithme est donc plus efficace.*

Limites et diagnostic. Une chaîne est initialisée aléatoirement. Il faut donc attendre un peu avant d'obtenir des échantillons exploitables. On ajoute donc un paramètre `warmups` qui indique le nombre d'itérations à réaliser avant de commencer à collecter les échantillons. Une solution classique pour vérifier qu'une chaîne a convergé est de comparer les résultats à ceux d'autres chaînes. L'indicateur $\hat{R}$ compare ainsi la moyenne des variances de chaque chaîne à la variance totale (sur les échantillons de toutes les chaînes) [Gelman et Rubin, 1992]. Si toutes les chaînes ont convergé vers la même distribution, le rapport entre ces deux quantités $\hat{R}$ est 1 [Stan Development Team, 2024, Section 16.3].

L'algorithme de Metropolis-Hastings a tendance à produire des échantillons artificiellement corrélés. En cas de rejet, deux échantillons successifs sont même égaux. Pour atténuer ce problème on ajoute un paramètre `thinning` qui indique le nombre d'étapes à ignorer entre deux échantillons (si `thinning = n`, on ne garde qu'un échantillon sur `n`). La taille effective de l'échantillon (*Effective Sample Size* ESS) permet de mesurer le nombre d'échantillons utiles pour représenter la distribution [Stan Development Team, 2024, Section 16.4].

Exemple. *Sur l'exemple* `gauss`*, pour 4 chaînes avec 1000 échantillons et 1000 étapes d'initialisation (*`num_samples=1000, warmups=1000`*) on obtient par exemple les diagnostics suivants :*

	$\hat{R}$		*ESS*	
	x	y	x	y
`SimpleMetropolis`	*1.93*	*1.98*	*21*	*8*
`MetropolisHastings`	*1.15*	*1.10*	*66*	*43*

Ces résultats dépendent bien sûr de l'exécution, mais ils confirment que `SimpleMetropolis` *converge plus difficilement que* `MetropolisHastings` *et génère moins d'échantillons exploitables.*

```
class MCMC(ImportanceSampling):
    def __init__(self, num_samples):
        self.num_samples = num_samples
        self.score = 0 # current score
        self.cache = {} # samples cache
        self.x_samples = {} # samples store
        self.x_scores = {} # X scores

    def sample(self, dist, name: str):
        try:
            v = self.cache[name] # reuse if possible
        except KeyError:
            v = dist.sample() # otherwise draw a sample
        self.x_samples[name] = v # store the sample
        self.x_scores[name] = dist.log_prob(v) # log x score
        return v

    # observe, assume and factor inherited from ImportanceSampling

    def mh(self, p_state):
        p_score, _, p_x_scores = p_state
        alpha = np.log(len(p_x_scores)) - np.log(len(self.x_scores))
        alpha += self.score - p_score
        for x in self.cache.keys() & self.x_scores.keys():
            alpha += self.x_scores[x]
            alpha -= p_x_scores[x]
        return np.exp(alpha)

    def infer(self, model, *args):
        samples = []
        new_value = model(*args, **kwargs) # generate first trace
        for _ in range(self.num_samples):
            p_state = self.score, self.x_samples, self.x_scores # store the state
            p_value = new_value # store current value
            regen = np.random.choice([n for n in self.x_samples])
            self.cache = deepcopy(self.x_samples) # use samples as next cache
            del self.cache[regen] # force regen to be resampled
            self.score, self.x_samples, self.x_scores = 0, {}, {} # reset the state
            new_value = model(*args) # regen a new trace
            alpha = self.mh(p_state)
            u = np.random.random()
            if not (u < alpha):
                self.score, self.x_samples, self.scores = p_state # rollback
                new_value = p_value
            samples.append(new_value)
        return Empirical(samples)
```

FIGURE 6 – Metropolis-Hastings

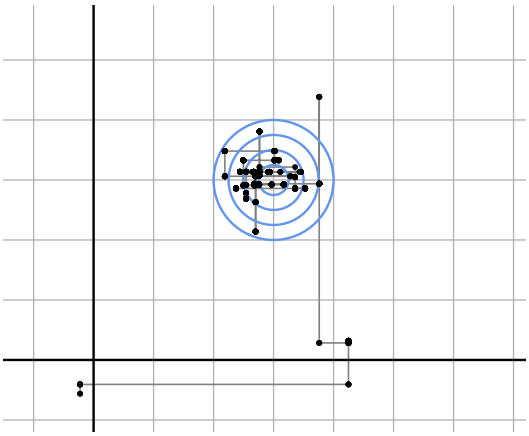


FIGURE 7 – Estimation d'une position à partir d'observations bruitées avec 1000 itérations de `MetropolisHastings`.

3 Sémantique

La sémantique est un domaine de l'informatique théorique dont le but est de donner un sens mathématique aux types et aux programmes indépendamment de leur syntaxe. On peut ainsi raisonner sur la sémantique d'un programme pour en prouver des propriétés (terminaison, correction, équivalence avec d'autres programmes…). On peut aussi utiliser la sémantique comme guide pour concevoir des langages de programmation dont les constructions reflètent des structures qui apparaissent dans les modèles mathématiques.

En programmation probabiliste, la sémantique d'un programme représente une loi de probabilité. Le type de retour d'un programme est interprété par un ensemble de valeurs de retours possibles. Un programme est interprété par une distribution de probabilité sur ces valeurs de retour.
Par exemple, le programme `coin` page 8 représente la loi du « biais de la pièce étant donnée l'observation des tirages indépendants », il est de type `float` interprété par l'ensemble des réels $\mathbb{R}$ et `infer(coin)` est interprété par la distribution de probabilité Bêta dont la densité est $\texttt{Beta}(\texttt{\#pile}+1, \texttt{\#face}+1)$.

Il existe plusieurs façons de calculer la distribution associée à un programme probabiliste. Dans cette partie, nous en présenterons deux que l'on retrouve dès les premiers travaux en sémantique des langages probabilistes [Kozen, 1979]. La *sémantique par noyau* qui interprète les programmes par des mesures paramétrées est présentée Section 3.3. La *sémantique par densité* qui interprète les programmes comme des fonctions de densité est présentée Section 3.4.

Nous commençons par donner quelques éléments de théorie de la mesure qui sevent de référence pour les notations et les définitions. Nous présentons ensuite en Sec-

tion 3.2 un sous-ensemble simplifié de μ-PPL qui nous permet de définir les sémantiques par noyau et par densité. Nous terminons en Section 3.6 par une ouverture sur la sémantique des langages de programmation probabiliste d'ordre supérieur, qui prennent en argument des programmes de type fonctionnel. Nous montrons les limites des sémantiques où les programmes sont interprétés par des distributions de probabilité et qui ne traitent pas bien l'ordre supérieur.

3.1 Éléments de théorie de la mesure

Étant donné Ω un ensemble non vide, une *tribu* Σ est un ensemble de parties de Ω, contenant Ω, stable par complémentaires et unions dénombrables. Un *espace mesurable* (Ω, Σ) est la donnée d'un ensemble muni d'une tribu. Un *ensemble mesurable* est une partie de Ω appartenant à la tribu Σ. La *tribu engendrée* par un ensemble B de parties de Ω est la plus petite tribu contenant B. On dit que B est une base de la tribu. La tribu borélienne de $\mathbb{R}$, notée $\mathcal{B}(\mathbb{R})$ est engendrée par les intervalles ouverts. La tribu borélienne de $\mathbb{R}^n$ est engendrée par les produits de n intervalles ouverts.

Une *mesure* sur (Ω, Σ) est une fonction $\mu : \Sigma \to \mathbb{R}^+$ qui est σ-additive, *i.e.* pour toute famille dénombrable $(U_i)_{i\in\mathbb{N}}$ d'ensembles mesurables, $\mu(\biguplus_i U_i) = \sum_i \mu(U_i)$. La mesure de comptage sur un ensemble dénombrable Ω muni de la tribu discrète de toutes les parties de Ω associe à toute partie son cardinal. On dit que μ est une mesure de probabilité lorsque de plus, $\mu(\Omega) = 1$. Une mesure sur un espace mesurable engendré par une base B qui est stable par intersections finies [4] est uniquement définie par son image sur les ensembles mesurables de base. La mesure de Lebesgue sur $\mathbb{R}$ est l'unique mesure $\lambda : B(\mathbb{R}) \to \mathbb{R}^+$ qui associe à tout intervalle sa longueur. De même, la mesure de Lebesgue λ_n sur $\mathbb{R}^n$ est l'unique mesure qui associe à tout produit d'intervalle le produit de ses longueurs. L'ensemble des mesures $\mathbf{Meas}(\Omega, \Sigma)$ muni de la tribu engendrée par $\{\{\mu \mid \mu(U) < r\}, \forall U \in \Sigma\ \forall r \in [0,1]\}$ est un espace mesurable.

Une *fonction mesurable* $f : (\Omega_1, \Sigma_1) \to (\Omega_2, \Sigma_2)$ est une fonction $f : \Omega_1 \to \Omega_2$ telle que $\forall V \in \Sigma_2,\ f^{-1}(V) \in \Sigma_1$. Étant donnés μ une mesure sur (Ω_1, Σ_1) et une fonction mesurable $f : (\Omega_1, \Sigma_1) \to (\Omega_2, \Sigma_2)$, la mesure *image* $f_*\mu$ sur (Ω_2, Σ_2) est définie par : $f_*\mu(V) = \mu(f^{-1}(V))$.

Un *noyau* $k : (\Omega_1, \Sigma_1) \rightsquigarrow (\Omega_2, \Sigma_2)$ est une fonction $k : \Omega_1 \times \Sigma_2 \to \mathbb{R}^+$ telle que pour tout $x \in \Omega_1$, $k(x, -) : \Sigma_2 \to \mathbb{R}^+$ est une mesure et pour toute partie mesurable $V \in \Sigma_2$, $k(-, V) : \Omega_1 \to \mathbb{R}^+$ est une fonction mesurable. On peut penser à un noyau comme une mesure sur (Ω_2, Σ_2) paramétrée de façon mesurable par les éléments Ω_1.

Pour toute mesure μ sur (Ω, Σ) et pour toute fonction mesurable $f : (\Omega, \Sigma) \to (\mathbb{R}, \mathcal{B}(\mathbb{R}))$, l'espérance de f par rapport à μ est l'intégrale de Lebesgue $\int f(x)\mu(dx)$. L'opérateur qui à associe à une fonction mesurable son espérance est appelé l'*opérateur d'intégration* induit par la mesure μ. Deux mesures sont égales si elles induisent le même opérateur d'intégration.

Une *variable aléatoire réelle* X sur un espace probabiliste $(\Omega, \Sigma, \mathbb{P})$ est une fonction mesurable $X : (\Omega, \Sigma) \to (\mathbb{R}^n, \mathcal{B}(\mathbb{R}))$. La loi d'une variable aléatoire X est la mesure

4. On parle de π-systèmes, voir [Billingsley, 1986, Section 3]

de probabilité sur $\mathbb{R}^n$ définie par la mesure image $X_*\mathbb{P}$, *i.e.* $\mu(U) = \mathbb{P}(X^{-1}(U))$. Une variable aléatoire réelle X est dite *à densité* par rapport à une mesure μ s'il existe une fonction de densité $f : \mathbb{R}^n \to \mathbb{R}^+$ telle que pour tout mesurable U, $\mathbb{P}(X \in U) = \int_U f d\mu$ Une variable aléatoire est *discrète* lorsqu'elle prend ses valeurs dans un ensemble au plus dénombrable (par exemple les distributions Bernoulli, binomiale, uniforme ou catégorique). Les variables aléatoires discrètes sont à densité par rapport à la mesure de comptage :

$$\mathbb{P}(X = x) = \sum_{\omega} \#\{\omega \mid X(\omega) = x\}$$

Une variable aléatoire est *continue* lorsqu'elle est à densité par rapport à la mesure de Lebesgues, sa fonction de répartition $F(x) = \mathbb{P}(\{\omega \mid X(\omega) \leq x\})$ est donnée par : $F(x) = \mathbb{P}(X \leq x) = \int_{-\infty}^{x} f(u)\lambda(du)$ (par exemple les distributions uniforme, bêta ou gaussienne). Les lois usuelles et leurs caractéristiques (support, densité, espérance) sont décrites en Figure 9.

Le *produit* $A_1 \otimes A_2$ de deux espaces mesurables $A_1 = (\Omega_1, \Sigma_1)$ et $A_2 = (\Omega_2, \Sigma_2)$ est donné par le produit cartésien d'ensembles $\Omega_1 \times \Omega_2$ muni de la tribu produit notée $\Sigma_1 \otimes \Sigma_2$ engendrée par les pavés $U_1 \times U_2$ pour $U_i \in \Sigma_i$. Étant données deux mesures μ_1 et μ_2 définies respectivement sur A_1 et A_2, la mesure produit $\mu_1 \otimes \mu_2$ sur $A_1 \otimes A_2$ est définie sur les mesurables de base $U_1 \times U_2$ par $\mu_1 \otimes \mu_2(U_1 \times U_2) = \mu_1(U)\,\mu_2(U_2)$. Étant donnés deux noyaux $k_1 : A_1 \rightsquigarrow B_1$ et $k_2 : A_2 \rightsquigarrow B_2$, leur produit tensoriel $k_1 \otimes k_2 : A_1 \otimes A_2 \rightsquigarrow B_1 \otimes B_2$ est le noyau défini sur les éléments de base par : $k_1 \otimes k_2((x_1, x_2), (U_1 \otimes U_2)) = k_1(x_1, U_1)\, k_2(x_2, U_2)$.

Le *cube infini* $[0,1]^{\mathbb{N}}$ est le produit dénombrable de $[0,1]$, muni de la tribu de Lebesgue, c'est-à-dire la plus grande tribu telle que les projections sont mesurables. On définit la mesure uniforme ρ sur le cube continu par extension de Kolmogorov [Billingsley, 1986, Theorem 36.1]. Cette mesure est telle que pour toute partie finie $F \subset \mathbb{N}$, la mesure image de ρ le long des projections $\pi_F : [0,1]^{\mathbb{N}} \to [0,1]^F$ sur les coordonnées indicées dans F est la mesure de Lebesgue sur $[0,1]^F$, $\lambda_F = \pi_{F*}\rho$. Pour toute fonction mesurable $g : [0,1]^F \to V$, on a la formule de changement de variable :

$$\int g(\pi_F(R)\rho(dR)) = \int g(R_F)\lambda_F(dR_F)$$

3.2 Syntaxe

La syntaxe d'un sous-ensemble simplifié de μ-PPL est présentée en Figure 10. Pour simplifier la présentation de la sémantique, cette syntaxe impose une forme normale administrative (*A normal form* ANF) [Flanagan *et al.*, 1993] : le résultat d'un appel de fonction est toujours capturé par une variable. Cette convention ne change rien à l'expressivité du langage. Il est toujours possible d'introduire des variables intermédiaires pour capturer le résultat d'un calcul.

Les types de données possibles sont le type vide (`None`), le type des booléens (`bool`), des entiers (`int`), des réels (`real`), des tuples ($\mathtt{t}_1 \times \mathtt{t}_2$) et des distributions sur des valeurs de type `t` (`dist(t)`). Les seules expressions sont les valeurs élémentaires : une constante (c), une variable (x), une paire de valeurs ((e_1, e_2)) ou l'application d'un

Loi	Paramètre	Support	Densité $\mathbb{P}(X=k)$	Fonction de répartitioni $F_X(x)$	Espérance	Variance
Uniforme (discrète)	$n \in \mathbb{N}^*$	{1,. . .,n}	$\frac{1}{n}$	$\frac{x}{n}$	$\frac{n+1}{2}$	$\frac{n^2-1}{12}$
Bernoulli	$p \in]0,1[$	$\{0,1\}$	$\begin{cases} p & \text{si } k=1 \\ 1-p & \text{si } k=0 \end{cases}$	$\begin{cases} 1-p & \text{si } x=0 \\ p & \text{si } x=1 \end{cases}$	p	$p(p-1)$
Binomiale	$n \in \mathbb{N}^*$ $p \in]0,1[$	$\{0,\ldots,n\}$	$\binom{n}{k} p^k (1-p)^{n-k}$	$\sum_{k=0}^{x} \binom{n}{k} p^k (1-p)^{n-k}$	np	$np(1-p)$
Géométrique	$p \in]0,1[$	$\mathbb{N}^*$	$(1-p)^{k-1}p$	$1-(1-p)^x$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
Poisson	$\mu > 0$	$\mathbb{N}$	$e^{-\mu}\frac{\mu^k}{k!}$	$\sum_{k=0}^{x} e^{-\mu}\frac{\mu^k}{k!}$	μ	μ

FIGURE 8 – Tableau des lois discrètes classiques.

Loi	Paramètre	Support	Densité $f_X(x)$	Fonction de répartitioni $F_X(x)$	Espérance	Variance
Uniforme (continue)	$a < b$	$[a, b[$	$\frac{1}{b-a}$	$\frac{x-a}{b-a}$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
Normale	$m \in \mathbb{R}$ $\sigma > 0$	$\mathbb{R}$	$\frac{1}{\sqrt{(2\pi\sigma^2)}}\, exp\left(-\frac{(x-m^2)}{2\sigma^2}\right)$	$\int_{-\infty}^{x} f_X(y)dy$	m	σ^2
Exponentielle	$\lambda > 0$	$[0, \infty[$	$\lambda e^{-\lambda x}$	$1 - e^{-\lambda x}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
Bêta	$\alpha > 0$ $\beta > 0$	$]0, 1[$	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$	$\int_{-\infty}^{x} \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$	$\frac{\alpha}{\alpha+\beta}$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$

FIGURE 9 – Tableau des lois continues classiques.

$$t ::= \texttt{None} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{real} \mid t \times t \mid \texttt{dist}(t)$$

$$e ::= c \mid x \mid (e, e) \mid op(e)$$

$$s ::= \texttt{pass} \mid x = e \mid x = f(e) \mid \texttt{if } e\texttt{: } s \texttt{ else: } s \mid s; s \\ \mid x = \texttt{sample}(e) \mid \texttt{factor}(e)$$

$$d ::= \texttt{def } f(x)\texttt{: } s \texttt{ return } e \mid d\ d$$

FIGURE 10 – Syntaxe de μ-PPL

opérateur primitif ($op(e)$ par exemple `+`, `-` ou `*`). On retrouve ensuite les commandes classiques pour un langage impératif : ne rien faire (`pass`), déclarer une variable ($x = e$, $x = f(e)$), effectuer un test (`if` e`:` s_1 `else:` s_2) ou enchaîner deux commandes (s_1 ; s_2). On ajoute les commandes probabilistes (x = `sample`(e) et `factor`(e)). Comme dans la Section 2, les opérateurs `observe` et `assume` peuvent être définis à partir de `factor` et n'apparaissent pas dans le noyau.

```
def assume(p):
    if p: factor(0)
    else: factor(-np.inf) # zero probability

def observe(d, v):
    factor(d.log_prob(v)) # log–likelihood of v w.r.t. d
```

Enfin, on peut déclarer des fonctions (`def` $f(x)$`:` s `return` e). Chaque fonction est caractérisée par un paramètre (une variable x), le corps de la fonction (une commande s), et une valeur de retour (une expression e).

3.3 Sémantique par noyau

La sémantique par noyau est une interprétation classique des programmes probabilistes [Kozen, 1979, Staton, 2017]. Un modèle décrit une mesure sur les valeurs de sorties possibles. Les types sont interprétés par des espaces mesurables. Les expressions sont interprétées par des fonctions mesurables, et les commandes sont interprétées par des noyaux qui associent à un environnement initial une mesure sur les environnements obtenus après l'exécution de la commande.

Types et environnements. L'interprétation des types de μ-PPL par des espaces mesurables est présentée en Figure 11. L'interprétation des types de base est intuitive[5]. L'interprétation du type produit et du type des distributions est définie récursivement à l'aide de l'interprétation des sous-types (voir les définitions des espaces mesurables sur

5. Pour alléger les notations on note aussi t l'ensemble des valeurs de type t.

$[\![t]\!]$	$=$	$t,\ \Sigma_t$	*Espace mesurable*
$[\![\texttt{None}]\!]$	$=$	$\{*\},\ \mathcal{P}(\{*\})$	*l'espace discret à un seul point*
$[\![\texttt{bool}]\!]$	$=$	$\mathbb{B},\ \mathcal{P}(\mathbb{B})$	*l'espace discret à deux points* `true` *et* `false`
$[\![\texttt{int}]\!]$	$=$	$\mathbb{N},\ \mathcal{P}(\mathbb{N})$	*l'espace discret des entiers*
$[\![\texttt{real}]\!]$	$=$	$\mathbb{R},\ \mathcal{B}(\mathbb{R})$	*l'espace Borélien sur les réels*
$[\![t_1 \times t_2]\!]$	$=$	$[\![t_1]\!] \otimes [\![t_2]\!]$	*l'espace produit* $t_1 \times t_2$, $\Sigma_{t_1} \otimes \Sigma_{t_2}$
$[\![\texttt{dist}(t)]\!]$	$=$	$\mathbf{Meas}([\![t]\!])$	*l'espace mesurable des mesures sur* $[\![t]\!]$

FIGURE 11 – Sémantique des types de μ-PPL

l'ensemble des mesures et sur le produit d'espaces mesurables données en Section 3.1). Soit $\mathcal{V}$ un ensemble dénombrable de variables typées. On note Γ l'ensemble des environnements γ qui associent une valeur $\gamma(x)$ à toute variable x de type t. Γ est l'espace mesurable produit $\bigotimes_{x:t\in\mathcal{V}} [\![t]\!]$.

Le langage que nous considérons est du premier ordre : les fonctions ne peuvent pas prendre de fonction en paramètre. Il n'y a donc pas de type fonctionnel $t \to t'$, mais nous autorisons les définitions de fonctions à travers un environnement de fonctions séparé de l'environnement des variables. Soit $\mathcal{F}$ un ensemble dénombrable de noms de fonctions. On note Φ l'ensemble des environnements de fonctions ϕ associant un noyau $\phi(f) : [\![t]\!] \rightsquigarrow [\![t']\!]$ à toute fonction $f \in \mathcal{F}$ de paramètre x de type t et de type de retour t'.

Définitions des noyaux. La sémantique par noyau de μ-PPL est présentée en Figure 12. L'interprétation d'un programme est décrite par trois règles, une pour les expressions e, une pour les commandes s et une pour les déclarations d.

$[\![e]\!]^\phi : \Gamma \to t$: Une expression de type t est interprétée par une fonction mesurable qui associe à tout environnement $\gamma \in \Gamma$ une valeur de type t. L'interprétation d'une variable x renvoie la valeur $\gamma(x)$ stockée dans l'environnement γ. L'interprétation d'une expression composée, (e_1, e_2) ou $op(e)$, construit récursivement la valeur de retour en interprétant chaque sous-expression.

$[\![s]\!]^\phi : \Gamma \rightsquigarrow \Gamma$: Une commande est interprétée par un noyau vu comme une mesure $[\![s]\!]^\phi_\gamma \in \mathbf{Meas}(\Gamma)$ sur les environnements, paramétrée par l'environnement γ. La commande `pass` ne change pas l'environnement et renvoie donc la mesure de Dirac δ_γ [6] sur l'environnement courant γ. La commande $x = e$ renvoie la mesure de Dirac sur l'environnement augmenté de la définition de x. La commande `if` e`:` s_1 `else:` s_2 exécute la commande s_1 ou la commande s_2 en fonction du résultat de l'évaluation de la condition. La commande $x =$ `sample`(e) renvoie une mesure sur tous les environnements possibles quand x parcourt la distribution définie par e. De même, l'appel de fonction $x = f(e)$ renvoie une mesure sur tous

6. $\delta_\gamma(U) = 1$ si $\gamma \in U$ et 0 sinon.

$$
\begin{array}{lcl}
[\![c]\!]^{\phi}_{\gamma} & = & c \\
[\![x]\!]^{\phi}_{\gamma} & = & \gamma(x) \\
[\![op(e)]\!]^{\phi}_{\gamma} & = & op([\![e]\!]^{\phi}_{\gamma}) \\
[\![(e_1\,,\, e_2)]\!]^{\phi}_{\gamma} & = & ([\![e_1]\!]^{\phi}_{\gamma}\,,[\![e_2]\!]^{\phi}_{\gamma})
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{pass}]\!]^{\phi}_{\gamma}(U) & = & \delta_{\gamma}(U) \\
[\![x = e]\!]^{\phi}_{\gamma}(U) & = & \delta_{\gamma+[x\leftarrow[\![e]\!]^{\phi}_{\gamma}]}(U) \\
[\![\texttt{if}\ e\texttt{:}\ s_1\ \texttt{else}\texttt{:}\ s_2]\!]^{\phi}_{\gamma} & = & [\![s_1]\!]^{\phi}_{\gamma}(U)\ si\ [\![e]\!]^{\phi}_{\gamma}\ sinon\ [\![s_2]\!]^{\phi}_{\gamma}(U) \\
[\![x = \texttt{sample}(e)]\!]^{\phi}_{\gamma}(U) & = & \displaystyle\int [\![e]\!]^{\phi}_{\gamma}(dv)\ \delta_{\gamma+[x\leftarrow v]}(U) \\
[\![x = f(e)]\!]^{\phi}_{\gamma}(U) & = & \displaystyle\int \mu(dv)\ \delta_{\gamma+[x\leftarrow v]}(U) \\
 & & avec\ \mu = \phi(f)([\![e]\!]^{\phi}_{\gamma}) \\
[\![\texttt{factor}(e)]\!]^{\phi}_{\gamma}(U) & = & [\![e]\!]^{\phi}_{\gamma}\ \delta_{\gamma}(U) \\
[\![s_1\,;\, s_2]\!]^{\phi}_{\gamma}(U) & = & \displaystyle\int [\![s_1]\!]^{\phi}_{\gamma}(d\gamma_1)\ [\![s_2]\!]^{\phi}_{\gamma_1}(U)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{def}\ f(x)\texttt{:}\ s\ \texttt{return}\ e]\!]^{\phi} & = & \phi + [f \leftarrow \lambda v.\ \lambda U.\ k(v,U)] \\
 & & avec\ k(v,U) = \displaystyle\int [\![s]\!]^{\phi}_{[x\leftarrow v]}(d\gamma)\ \delta_{[\![e]\!]^{\phi}_{\gamma}}(U) \\
[\![d_1\ d_2]\!]^{\phi} & = & [\![d_2]\!]^{\phi_1}\ \ avec\ \phi_1 = [\![d_1]\!]^{\phi}
\end{array}
$$

$$
\begin{array}{lcl}
[\![\texttt{infer}(f\,,\, e)]\!]^{\phi}_{\gamma}(U) & = & \begin{cases} \dfrac{\mu(U)}{\mu(\top)} & si\ 0 < \mu(\top) < \infty \\ \text{Erreur} & sinon \end{cases} \\
 & & avec\ \mu(U) = \phi(f)([\![e]\!]^{\phi}_{\gamma})
\end{array}
$$

FIGURE 12 – Sémantique par noyau de μ-PPL

les environnements possibles quand x parcourt la mesure définie par $f(e)$. La commande `factor(`e`)` ajoute un facteur multiplicatif à la mesure courante. Pour enchaîner deux commandes s_1 ; s_2 on intègre l'interprétation de s_2 sur tous les environnements possibles après s_1.

$[\![d]\!] : \Phi \to \Phi$: La sémantique d'une déclaration est interprétée par une fonction qui associe à tout environnement de fonctions un nouvel environnement de fonctions. L'interprétation d'une déclaration `def` f`(`x`):` s `return` e ajoute l'interprétation de f à l'environnement courant. L'interprétation de f est un noyau $[\![t]\!] \rightsquigarrow [\![t']\!]$ vu comme une mesure sur les valeurs du type t' de l'expression de sortie e paramétrée par la valeur v du paramètre x de type t. Le corps de la fonction est exécuté dans un environnement initial qui ne contient que l'argument $[x \leftarrow v]$ pour obtenir une mesure sur les environnements. On intègre ensuite l'interprétation de l'expression de sortie e sur tous les environnements possibles.

Inférence. La sémantique de `infer(`f`,` e`)` calcule la mesure définie par le modèle $f(e)$. L'opérateur de conditionnement `factor` permet de manipuler le score de certaines exécutions. La mesure obtenue n'est donc pas forcément normalisée : le score total sur toutes les exécutions possibles n'est pas nécessairement 1. On re-normalise donc la mesure obtenue pour obtenir une distribution sur les valeurs de retour (on note $\top$ l'ensemble des sorties possibles de type t). Il est possible que la normalisation échoue si $\mu(\top)$ est nul ou infini. On renvoie alors une erreur. En pratique, l'inférence ne fonctionnera pas sur de tels modèles et c'est à l'utilisateur d'éviter cette situation.

Exemple. *On peut d'abord vérifier que la construction* `sample` *a le comportement attendu.*

```
def my_gaussian(mu, sigma):
  x = sample(Gaussian(mu, sigma))
  return x
```

On note ϕ l'environnement de fonctions qui contient la définition de `my_gaussian` *et $\gamma_0 = [\texttt{mu} \leftarrow m, \texttt{sigma} \leftarrow s]$ l'environnement initial qui contient la définition des paramètres.*

$$
\begin{aligned}
&\phi(\texttt{my\_gaussian})(m,s)(U) \\
&\quad = \int [\![\texttt{x = sample(Gaussianm, sigma))}]\!]_{\gamma_0}\,(d\gamma)\;\delta_{[\![x]\!]_\gamma}(U) \\
&\quad = \int\int \mathcal{N}(m,s)(dv)\;\delta_{\gamma_0+[\texttt{x}\leftarrow v]}(d\gamma)\;\delta_{\gamma(x)}(U) \\
&\quad = \int \mathcal{N}(m,s)(dv)\;\delta_v(U) \\
&\quad = \int_U \frac{1}{s\sqrt{2\pi}} e^{\frac{-(x-m)^2}{2s^2}}\,dx \\
&\quad = \mathcal{N}(m,s)(U)
\end{aligned}
$$

En l'absence de conditionnement, la sémantique de `my_gaussian` *est directement une distribution. La normalisation ne change rien et on a donc :*

$$[\![\texttt{infer(my\_gaussian, (mu, sigma))}]\!]^{\phi}_{\gamma} = \mathcal{N}(m,s)$$

L'exemple suivant illustre le comportement de `observe`.

```
def beta_bernoulli(x):
  z = sample(Beta(a, b))
  observe(Bernoulli(z), x)
  return z
```

On note ϕ l'environnement de fonctions qui contient la définition de `beta_bernoulli` *et* $\gamma = [\texttt{a} \leftarrow a, \texttt{b} \leftarrow b, \texttt{x} \leftarrow x]$ *l'environnement initial.*

$$
\begin{aligned}
&\phi(\texttt{beta\_bernoulli})(x)(U) \\
&\quad = \int [\![\texttt{z = sample(Beta(a, b))}]\!]_{\gamma}(d\gamma_z) \\
&\qquad\quad \int [\![\texttt{observe(Bernoulli(z), x)}]\!]_{\gamma_z}(d\gamma_x)\ [\![z]\!]_{\gamma_x}(U) \\
&\quad = \int Beta(a,b)(dz)\delta_{\gamma+[\texttt{z}\leftarrow z]}(d\gamma_z) \\
&\qquad\quad \int pdf(Bernoulli(\gamma_z(z)), \gamma_z(x))\ \delta_{\gamma_z}(d\gamma_x)\ \delta_{\gamma_x(z)}(U) \\
&\quad = \int_U Beta(a,b)(dz)\ z^x(1-z)^{1-x} \\
&\quad \propto \int_U z^{a+x-1}(1-z)^{b+(1-x)-1}dz
\end{aligned}
$$

Cette mesure n'est pas normalisée, mais on reconnaît la fonction de densité de la distribution $Beta(a+x, b+(1-x))$ à une constante près. On en déduit la distribution de probabilité après re-normalisation :

$$[\![\texttt{infer(beta\_bernoulli, x)}]\!]^{\phi}_{\gamma} = Beta(a+x, b+(1-x))$$

C'est un résultat classique. Les distributions Bêta et Bernoulli sont dites conjuguées [Fink, 1997]. La distribution a posteriori *est de la même famille (Bêta) que la distribution* a priori.

On peut également utiliser la sémantique par noyau pour prouver certaines équivalences de programmes. Par exemple :

`# mu defined on [a, b]` `x = sample(mu)`	$\equiv$	`x = sample(Uniform(a, b))` `observe(mu, x)`

`def coin():`
`  x = sample Bernoulli(0.5)`
`  return x`
	$\equiv$	`def coin():`
		`  x = sample(Gaussian(0, 1))`
		`  return x > 0`

`factor(a); ... ; factor(b)`	$\equiv$	`... ; factor(a * b)`
`x = sample(dx); y = sample(dy)`	$\equiv$	`y = sample(dy); x = sample(dx)`

3.4 Sémantique par densité

Certains algorithmes d'inférence avancés nécessitent de connaître la fonction de densité d'un modèle. C'est le cas de l'algorithme de Monte Carlo hamiltonien (HMC) [Betancourt, 2018] ou de l'inférence variationnelle (SVI) [Blei *et al.*, 2016].

Considérons un modèle probabiliste sans argument qui renvoie des valeurs de type t. La sémantique par densité interprète un modèle comme une fonction $f : P \to t \times \mathbb{R}^+$. Cette fonction associe à toute *exécution* du modèle, c'est-à-dire la donnée d'un échantillon p pour toutes les variables aléatoires, la valeur de retour $v(p)$ de type t et un score $W(p) \geq 0$. Cette fonction est déterministe : « *étant donné un oracle nous donnant un échantillon pour chaque variable aléatoire, on sait déterminer la valeur de retour du modèle et son poids* ».

Ainsi, pour toute valeur v et score w, on peut reconstruire la mesure μ sur les sorties possibles en intégrant sur toutes les exécutions possibles. Si ρ est une distribution de probabilité uniforme sur les exécutions p, alors :

$$\mu(U) = \int \rho(dp) W(p)\ \delta_{v(p)}(U)$$

Attention, en μ-PPL l'ensemble des variables aléatoires peut changer d'une exécution à l'autre et les définitions peuvent dépendre du contexte. C'est le cas dans l'exemple suivant.

```
def add_noise(obs, error):
  if error: x = sample(Gaussian(0, 100))
  else: x = sample(Gaussian(obs, 1))
  return x
```

Pour résoudre ce problème, nous suivons l'approche suivie par PyMC, Pyro ou μ-PPL et formalisée par [Lee *et al.*, 2020]. On associe un *nom* unique à chaque simulation d'une variable aléatoire, c'est-à-dire à chaque commande x = `sample`$(d, name)$.

Espace mesurable des oracles et probabilité sur les oracles. Notons Str l'ensemble infini dénombrable des noms possibles. Nous appelons *oracle* une fonction r qui associe à chaque commande x = `sample`$(d, name)$ une valeur aléatoire $r(name) \in [0, 1]$. On peut alors convertir cette valeur aléatoire en un échantillon v de la distribution d en utilisant l'inverse généralisée de sa fonction de répartition $v = icdf(d, r(name))$. C'est la méthode de la transformée inverse [Devroye, 2006]. L'ensemble des oracles $R = [0, 1]^{Str}$ est isomorphe au cube infini $[0, 1]^{\mathbb{N}}$. On le muni de la mesure uniforme que l'on note ρ (voir Section 3.1).

Définition de la densité La sémantique par densité de μ-PPL est présentée en Figure 13. Cette interprétation est très proche de la sémantique classique d'un langage impératif. Les seules différences sont la gestion de l'oracle et du score.

$(\!|e|\!)^{\phi} : \Gamma \to t$: La sémantique d'une expression de type t est identique à sa sémantique par noyau.

$(\!|s|\!)^{\phi} : \Gamma \times R \to \Gamma \times \mathbb{R}^+$: La sémantique d'une commande prend en paramètre un environnement et un oracle et renvoie un environnement associé à un score. La commande `pass` ne change pas l'environnement. Une définition de variable x = e ajoute la définition de x à l'environnement. La commande `if` e : s_1 `else` : s_2 exécute la commande s_1 ou la commande s_2 en fonction du résultat de l'évaluation de la condition. La commande x = `sample`(e, *name*) évalue l'expression e pour obtenir une distribution et renvoie l'échantillon $icdf((\!|e|\!)^{\phi}, r(name))$ associé à la valeur aléatoire $r(name)$ de l'oracle et un score de 1. L'appel de fonction x = f`(`e`)` évalue la fonction pour obtenir une valeur et un score. On ajoute alors la définition de x à l'environnement. La commande `factor`(e) ne renvoie pas de valeur. Elle renvoie un score qui correspond à l'évaluation de l'expression e. Pour enchaîner deux commandes s_1 ; s_2, on exécute s_2 dans l'environnement renvoyé par s_1 et on multiplie les scores de s_1 et de s_2.

$(\!|d|\!) : \Phi \to \Phi$: La sémantique d'une déclaration modifie un environnement de fonctions. L'interprétation d'une déclaration `def` f`(`x`)` : s `return` e ajoute la définition de f à l'environnement ϕ. La fonction prend en argument la valeur v du paramètre x et l'oracle r. Le corps de la fonction s est interprété dans un environnement initial qui ne contient que l'argument $[x \leftarrow v]$. On obtient ainsi un environnement γ et un score W que l'on utilise pour calculer la valeur de l'expression de sortie et renvoyer ce score.

Inférence. La sémantique de `infer`(f, e) calcule la mesure décrite par le modèle en intégrant la sémantique de $f(e)$ uniformément sur tous les oracles possibles, c'est-à-dire toutes les exécutions possibles du modèle. On re-normalise ensuite la mesure obtenue pour obtenir une *distribution* sur les valeurs de retour.

Exemple. *Pour illustrer cette sémantique, revenons sur les exemples de la section précédente. Quand cela est possible, on utilise le nom de variable comme identifiant dans l'oracle.*

$$\phi(\texttt{my\_gaussian})(m,s)(r) = icdf(\mathcal{N}(m,s), r(\texttt{x})), 1$$

$$\begin{aligned}
\mu(U) &= \int_0^1 1\ \delta_{icdf(\mathcal{N}(m,s),r_x)}(U)\ dr_x \\
&= \int \mathcal{N}(m,s)(dv)\ \delta_v(U) \qquad \text{avec } v = icdf(\mathcal{N}(m,s), r_x) \\
&= \mathcal{N}(m,s)(U)
\end{aligned}$$

Comme dans la section précédente on obtient donc la distribution $\mathcal{N}(m,s)$.

Pour illustrer le comportement de `observe`*, reprenons le second exemple.*

$$\begin{aligned}
\phi(\texttt{beta_bernoulli})(x)(r) &= (z, pdf(Bernoulli(z), x)) \\
&\text{avec } z = icdf(Beta(a,b), r(\texttt{z}))
\end{aligned}$$

$$
\begin{array}{lcl}
(\!|c|\!)^{\phi}_{\gamma} & = & c \\
(\!|x|\!)^{\phi}_{\gamma} & = & \gamma(x) \\
(\!|op(e)|\!)^{\phi}_{\gamma} & = & op((\!|e|\!)^{\phi}_{\gamma}) \\
(\!|(e_1,\, e_2)|\!)^{\phi}_{\gamma} & = & \left((\!|e_1|\!)^{\phi}_{\gamma}, (\!|e_2|\!)^{\phi}_{\gamma}\right) \\
 & & \\
(\!|\texttt{pass}|\!)^{\phi}_{\gamma}(r) & = & (\gamma, 1) \\
(\!|x = e|\!)^{\phi}_{\gamma}(r) & = & \left(\gamma + \left[x \leftarrow (\!|e|\!)^{\phi}_{\gamma}\right], 1\right) \\
(\!|\texttt{if}\ e\texttt{:}\ s_1\ \texttt{else}\texttt{:}\ s_2|\!)^{\phi}_{\gamma}(r) & = & (\!|s_1|\!)^{\phi}_{\gamma}(r)\ si\ (\!|e|\!)^{\phi}_{\gamma}\ sinon\ (\!|s_2|\!)^{\phi}_{\gamma}(r) \\
(\!|x = \texttt{sample}(e, name)|\!)^{\phi}_{\gamma}(r) & = & \left(\gamma + \left[x \leftarrow icdf\left((\!|e|\!)^{\phi}_{\gamma}, r(name)\right)\right], 1\right) \\
(\!|x = f(e)|\!)^{\phi}_{\gamma}(r) & = & (\gamma + [x \leftarrow v], W) \\
 & & avec\ v, W = \phi(f)((\!|e|\!)^{\phi}_{\gamma})(r) \\
(\!|\texttt{factor}(e)|\!)^{\phi}_{\gamma}(r) & = & \left(\gamma, (\!|e|\!)^{\phi}_{\gamma}\right) \\
(\!|s_1\texttt{;}\ s_2|\!)^{\phi}_{\gamma}(r) & = & (\gamma_2, W_1 \cdot W_2)\ avec\ (\gamma_1, W_1) = (\!|s_1|\!)^{\phi}_{\gamma}(r) \\
 & & \qquad\qquad et\ (\gamma_2, W_2) = (\!|s_2|\!)^{\phi}_{\gamma_1}(r) \\
 & & \\
(\!|\texttt{def}\ f(x)\texttt{:}\ s\ \texttt{return}\ e|\!)^{\phi} & = & \phi + \left[f \leftarrow \lambda v\ \lambda r.\ \left((\!|e|\!)^{\phi}_{\gamma}, W\right)\right. \\
 & & \qquad avec\ (\gamma, W) = (\!|s|\!)^{\phi}_{[x \leftarrow v]}(r)\Big] \\
(\!|d_1\ d_2|\!)^{\phi} & = & (\!|d_2|\!)^{\phi_1}\ avec\ \phi_1 = (\!|d_1|\!)^{\phi} \\
 & & \\
(\!|\texttt{infer}(f,\, e)|\!)^{\phi}_{\gamma}(U) & = & \begin{cases} \dfrac{\mu(U)}{\mu(\top)} & si\ 0 < \mu(\top) < \infty \\ \text{Erreur} & sinon \end{cases} \\
 & & avec\ \mu(U) = \displaystyle\int \rho(dr)\ W(r)\ \delta_{v(r)}(U) \\
 & & et\ v(r), W(r) = \phi(f)((\!|e|\!)^{\phi}_{\gamma}(r))
\end{array}
$$

FIGURE 13 – Sémantique par densité de μ-PPL

$$\begin{aligned}\mu(U) &= \int_0^1 pdf(Bernoulli(z), x)\ \delta_z(U)\ dr_z \qquad avec\ z = icdf(Beta(a,b), r_z)\\ &= \int pdf(Bernoulli(z), x)\ Beta(a,b)(dz)\delta_z(U)\\ &= \int_U Beta(a,b)(dz)\ z^x(1-z)^{1-x}\\ &\propto \int_U z^{a+x-1}(1-z)^{b+(1-x)-1}dz\end{aligned}$$

On retrouve également le résultat de la section précédente. Après normalisation on obtient donc la distribution $Beta(a+x, b+(1-x))$.

3.5 Correspondance des sémantiques par noyau et par densité

La proposition suivante montre que les deux sémantiques par noyau et par densité représentent la même distribution.

Proposition 3.1 (Équivalence sémantique)**.** *Soit* s *une commande, la sémantique par densité de* s *est la densité de la mesure calculée par la sémantique par noyau.*

$$[\![\mathsf{s}]\!]^{\phi}_{\gamma}(U) = \int \rho(dr) W_{\mathsf{s}}(r) \delta_{\gamma_{\mathsf{s}}(r)}(U) \qquad où\ \gamma_{\mathsf{s}}(r), W_{\mathsf{s}}(r) = (\!|\mathsf{s}|\!)^{\phi}_{\gamma}(r)$$

Démonstration. On raisonne par induction sur la structure des commandes, puis par cas.

`pass` et $x = e$: il suffit de dérouler les définitions. On conclut en utilisant que $\int \rho(dr) = 1$ car ρ est la mesure de probabilité uniforme sur les oracles. On conclut en rappelant que par définition, les sémantiques sur les expressions sont les mêmes $[\![e]\!]^{\phi}_{\gamma} = (\!|e|\!)^{\phi}_{\gamma}$.

`if` e`:` s_1 `else:` s_2 : on utilise l'égalité des sémantiques sur les expressions, puis on applique l'hypothèse de récurrence sur s_1 et sur s_2.

x = f`(`e`)` avec `def` f`(`x`):` s' `return` e' : on utilise l'égalité des sémantiques sur les expressions e et e', ainsi que l'hypothèse de récurrence appliquée à la sous-commande s'.

x = `sample(`e, *name*`)` : on a par définition,

$$[\![s]\!]^{\phi}_{\gamma}(U) = \int [\![e]\!]^{\phi}_{\gamma}(dv)\ \delta_{\gamma+[x\leftarrow v]}(U)$$
$$\int \rho(dr) W_s(r) \delta_{\gamma_s(r)}(U) = \int \rho(dr) \delta_{\gamma+\left[x\leftarrow icdf\left((\!|e|\!)^{\phi}_{\gamma}, r(name)\right)\right]}(U)$$

Notons $\pi_{name} : [0,1]^{Str} \to [0,1]$ la projection sur la composante d'indice *name* correspondant au nom unique associé par la commande x = `sample(`e, *name*`)`. On remarque que $r(name) = \pi_{name}(r)$ et la mesure image de l'uniforme le long de la projection $\pi_{name*}\rho = \lambda$ est la mesure de Lebesgue sur $[0,1]$. On applique le changement de variable $r_0 = \pi_{name}(r)$

$$\int \rho(dr) W_s(r) \delta_{\gamma_s(r)}(U) = \int \lambda(dr_0) \delta_{\gamma+\left[x\leftarrow icdf\left((\!|e|\!)^{\phi}_{\gamma}, r_0\right)\right]}(U)$$

puis le changement de variable $v = icdf(\llbracket e \rrbracket_\gamma^\phi, r_0)$, soit $r_0 = \llbracket e \rrbracket_\gamma^\phi(v)$

$$\int \rho(dr) W_s(r) \delta_{\gamma_s(r)}(U) = \int \llbracket e \rrbracket_\gamma^\phi(dv) \delta_{\gamma+[x \leftarrow v]}(U)$$

On conclut en utilisant l'égalité des sémantiques sur les expressions.

$s_1 ; s_2$: on utilise l'hypothèse d'induction sur les sous-commandes s_1 et s_2. On remarque ensuite que les espaces de noms Str_1 et Str_2 utilisés dans des opérateurs `sample` de s_1 et de s_2 respectivement, sont disjoints. On effectue les changements de variables le long des projections sur ces espaces de noms pour se ramener à l'hypothèse d'induction sur chacune des sous-commandes s_1 et s_2.

□

3.6 Ordre supérieur

Nous avons défini la syntaxe de μ-PPL avec soin pour en faire un langage du premier ordre dans lequel les programmes ne peuvent pas avoir des paramètres de type fonctionnel. C'est pour cette raison que nous avons séparé les environnements pour les paramètres et les environnements pour les définitions de fonctions. Cela nous a permis de définir la sémantique des programmes comme des noyaux. Cependant, la plupart des langages (dont python) permettent d'utiliser l'ordre supérieur. Malheureusement la sémantique intuitive des espaces mesurables et des noyaux ne convient pas à l'ordre supérieur comme le démontre le lemme d'Aumann (voir Proposition 3.2).

Plusieurs sémantiques ont été introduites à la fin des années 2020 pour donner un sens aux langages probabilistes fonctionnels (d'ordre supérieur) [Heunen *et al.*, 2017, Ehrhard *et al.*, 2018]. Ces deux modèles sont basés sur des formalisations différentes mais reposent sur les mêmes idées. Tout d'abord, les programmes sans paramètre, de type de base sont toujours interprétés par des mesures. Mais un type fonctionnel est interprété comme un ensemble de transformateurs de mesures. Cette interprétation est compatible avec l'intégration et la mesurabilité. Elle écarte le contre-exemple de Aumann en rendant l'évaluation compatible avec cette interprétation.

En 1961, Robert Aumann s'intéresse aux espaces de fonctions mesurables $\mathbf{Meas}(X, Y)$ entre espaces mesurables et montre que l'évaluation $\mathbf{eval} : \mathbf{Meas}(X, Y) \otimes X \to Y$ n'est pas, en général, une fonction mesurable [Aumann, 1961]. Nous présentons une version simplifiée de son argument. Nous pouvons alors conclure que les espaces mesurables et les fonctions mesurables ne peuvent pas être un modèle de la programmation fonctionnelle d'ordre supérieur.

Proposition 3.2 (Lemme d'Aumann [Aumann, 1961])**.** *Il existe X et Y des espaces mesurables tels que quelle que soit la tribu sur l'espace des fonctions mesurables $\boldsymbol{Meas}(X, Y)$, l'évaluation $\boldsymbol{eval} : \boldsymbol{Meas}(X, Y) \otimes X \to Y$ n'est pas mesurable.*

Démonstration. Raisonnons[7] par l'absurde et supposons que pour tous X et Y, il existe une tribu telle que $\mathbf{eval} : \mathbf{Meas}(X, Y) \otimes X \to Y$ est mesurable.

7. Merci à Ohad Kammar et Thomas Ehrhard pour leurs simplification de la preuve présentée ici.

Considérons $X = \mathbb{R}$ muni de la tribu $\mathcal{C}(Y)$ engendrée par unions et intersections dénombrables de parties dénombrables et co-dénombrables (dont les complémentaires sont dénombrables) et l'espace mesurable discret $Y = \{0, 1\}$. Notons $Z = \mathbb{R}$ muni de la tribu discrète $\mathcal{P}(\mathbb{R})$ (toutes les parties sont mesurables).

Soit $d : Z \otimes X \to Y$ la fonction diagonale définie par :

$$\begin{cases} d(z, x) = 1 & \text{si } z = x \\ d(z, x) = 0 & \text{si } z \neq x \end{cases}$$

Quelque soit la tribu sur l'espace des fonctions mesurables $\mathbf{Meas}(X, Y)$, la curryfiée $\mathbf{curry}(d) : Z \to \mathbf{Meas}(X, Y)$ est mesurable car l'inverse de tout mesurable est une partie de $\mathbb{R}$. Par composition, $d = \mathbf{eval} \circ (\mathbf{curry}(d) \otimes \mathrm{id}_Y)$ est donc mesurable. On en déduit que $\Delta = \{(z, x) \in \mathbb{R}^2 \mid z = x\} = d^{-1}(\{1\})$ est mesurable pour la tribu produit $\mathcal{P}(\mathbb{R}) \otimes \mathcal{C}(\mathbb{R})$ engendrée par unions et intersections dénombrables de pavés, produits de parties de $\mathbb{R}$ et de dénombrables ou co-dénombrables.

Remarquons que tous les mesurables W de $\mathcal{P}(\mathbb{R}) \otimes \mathcal{C}(\mathbb{R})$ et notamment Δ vérifient la propriété suivante :

$$\begin{aligned} &\text{Il existe } B \in \mathbb{R} \text{ dénombrable et non vide tel que :} \\ &\qquad \text{si } (z, x) \in W \text{ tel que } x \notin B, \text{ alors } \forall x' \notin B,\ (z, x') \in W. \end{aligned} \tag{1}$$

Il suffit de prouver cette propriété pour les mesurables de bases et de démontrer qu'elle est stable par union et intersection dénombrable. Supposons que Δ vérifie la propriété (1). Alors, comme B est dénombrable, il existe deux réels $x \neq x' \notin B$. Or $(x, x) \in \Delta$, donc $(x, x') \in \Delta$. Comme on a choisi $x \neq x'$, on aboutit à une contradiction puisque Δ est la diagonale.

Revenons sur la preuve de la propriété (1). Pour un mesurable de base $U \times B$ où B est dénombrable, la propriété est vérifiée pour B. Pour un mesurable de base $U \times C$ où C est co-dénombrable, on choisit $B = \mathbb{R} \backslash C$ et la propriété est vérifiée car il n'y a pas de $(z, x) \in U \times C$ tel que $x \notin B$. Soient $(W_i)_{i \in \mathbb{N}}$ une famille dénombrable de mesurables de $\mathcal{P}(\mathbb{R}) \otimes \mathcal{C}(\mathbb{R})$ vérifiant la propriété (1) et B_i les ensembles dénombrables associés. On choisit $B = \cup_{i \in \mathbb{N}} B_i$ qui est dénombrable.

— Soient $(z, x) \in \cap_{i \in \mathbb{N}} W_i$ avec $x \notin B$ et $x' \notin B$. Pour tout i, $x' \notin B_i$, donc $(z, x') \in W_i$. On a montré que $(z, x') \in \cap_{i \in \mathbb{N}} W_i$.

— Soient $(z, x) \in \cup_{i \in \mathbb{N}} W_i$ avec $x \notin B$ qui est dénombrable et $x' \notin B$. Il existe j tel que $(z, x) \in W_j$. Or $x \notin B_j$, comme $x' \notin B_j$, on a $(z, x') \in W_j$. On a montré que $(z, x') \in \cup_{i \in \mathbb{N}} W_i$.

□

Conclusion

Nous avons présenté dans ces notes les concepts fondamentaux de la programmation probabiliste : modélisation bayésienne, conception d'un langage probabiliste, implémentation du moteur d'inférence et deux sémantiques formelles (par noyau et par densité) pour raisonner sur les programmes. Pour conclure, voici plusieurs directions pour aller plus loin.

Méthodes d'inférence avancées. Les langages probabilistes modernes proposent des méthodes d'inférence avancées qui sont efficaces pour de larges classes de modèles. Le moteur d'inférence de Stan utilise un algorithme de Monte Carlo hamiltonien (*Hamiltonian Monte Carlo* HMC) [Betancourt, 2018] optimisé appelé NUTS (*No U-Turn Sampler*) [Hoffman et Gelman, 2014]. Cette méthode est une instance de l'algorithme de Metropolis-Hastings présenté en Section 2.3 qui utilise les équations de la dynamique hamiltonienne pour générer des candidats qui ont une très faible probabilité de rejet.

L'inférence variationnelle (*Stochastic Variational Inference* SVI) [Blei *et al.*, 2016] est un autre algorithme qui est au cœur du langage Pyro. Cet algorithme calcule la distribution décrite par un modèle en résolvant un problème d'optimisation. Étant donnée une famille de lois paramétrées, on cherche la valeur du paramètre qui minimise la différence entre le modèle et un membre de la famille. On utilise une descente de gradient pour trouver le meilleur paramètre possible. Définir une famille de lois adaptée à un modèle est une tâche complexe et de nombreuses méthodes sont développées pour synthétiser automatiquement ces familles à partir du modèle [Kucukelbir *et al.*, 2017, Webb *et al.*, 2019, Baudart et Mandel, 2021].

Deep probabilistic programming. Ces méthodes d'inférence avancée reposent sur des techniques de *différentiation automatique* qui sont devenues populaires avec l'explosion de l'apprentissage profond. Pyro est ainsi fondé sur PyTorch [Paszke *et al.*, 2019], NumPyro est une version de Pyro fondée sur Jax [Bradbury *et al.*, 2018], et Edward [Tran *et al.*, 2017] est un autre langage probabiliste fondé sur Tensorflow [Abadi *et al.*, 2016]. Cette infrastructure permet de mélanger réseaux de neurones et modèles probabilistes pour programmer des modèles complexes [Bingham *et al.*, 2019, Tran *et al.*, 2017].

Sémantique et ordre supérieur. La plupart des langages probabilistes permettent d'écrire des programmes d'ordre supérieur. Comme nous l'avons montré à la fin de la section sur la sémantique, la définition des modèles de ces langages est délicate. Le modèle des espaces quasi-boréliens (QBS) [Heunen *et al.*, 2017] est peut-être le plus utilisé pour étudier la sémantique des programmes probabilistes et pour prouver la correction des algorithmes d'inférence [Ścibior *et al.*, 2018]. Les cônes intégrables [Ehrhard *et al.*, 2018, Ehrhard et Geoffroy, 2022] sont également un modèle de l'ordre supérieur. Les programmes probabilistes y sont interprétés par des morphismes linéaires intégrables qui forment une autre généralisation des matrices stochastiques aux probabilités continues.

Références

[Abadi *et al.*, 2016] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P. A., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y. et ZHENG, X. (2016). Tensorflow : A system for large-scale machine learning. *In OSDI*.

[Aumann, 1961] AUMANN, R. J. (1961). Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614 – 630.

[Baudart et Mandel, 2021] BAUDART, G. et MANDEL, L. (2021). Automatic guide generation for stan via numpyro. *In PROBPROG*.

[Betancourt, 2018] BETANCOURT, M. (2018). A conceptual introduction to Hamiltonian Monte Carlo. *CoRR*, abs/1701.02434.

[Billingsley, 1986] BILLINGSLEY, P. (1986). *Probability and Measure*. John Wiley and Sons, second édition.

[Bingham *et al.*, 2019] BINGHAM, E., CHEN, J. P., JANKOWIAK, M., OBERMEYER, F., PRADHAN, N., KARALETSOS, T., SINGH, R., SZERLIP, P. A., HORSFALL, P. et GOODMAN, N. D. (2019). Pyro : Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20:28 :1–28 :6.

[Blei *et al.*, 2016] BLEI, D. M., KUCUKELBIR, A. et MCAULIFFE, J. D. (2016). Variational inference : A review for statisticians. *CoRR*, abs/1601.00670.

[Bradbury *et al.*, 2018] BRADBURY, J., FROSTIG, R., HAWKINS, P., JOHNSON, M. J., LEARY, C., MACLAURIN, D., NECULA, G., PASZKE, A., VANDERPLAS, J., WANDERMAN-MILNE, S. et ZHANG, Q. (2018). JAX : composable transformations of Python+NumPy programs. `http://github.com/google/jax`.

[Carpenter *et al.*, 2017] CARPENTER, B., GELMAN, A., HOFFMAN, M. D., LEE, D., GOODRICH, B., BETANCOURT, M., BRUBAKER, M., GUO, J., LI, P. et RIDDELL, A. (2017). Stan : A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–32.

[Cooper, 1990] COOPER, G. F. (1990). The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42(2-3):393–405.

[Cusumano-Towner *et al.*, 2019] CUSUMANO-TOWNER, M. F., SAAD, F. A., LEW, A. K. et MANSINGHKA, V. K. (2019). Gen : a general-purpose probabilistic programming system with programmable inference. *In PLDI*.

[Devroye, 2006] DEVROYE, L. (2006). Nonuniform random variate generation. *Handbooks in operations research and management science*, 13:83–121.

[Ehrhard et Geoffroy, 2022] EHRHARD, T. et GEOFFROY, G. (2022). Integration in cones. *CoRR*, abs/2212.02371.

[Ehrhard *et al.*, 2018] EHRHARD, T., PAGANI, M. et TASSON, C. (2018). Measurable cones and stable, measurable functions : a model for probabilistic higher-order programming. *Proceedings of the ACM on Programming Languages*, 2(POPL):59 :1–59 :28.

[Fink, 1997] FINK, D. (1997). A compendium of conjugate priors.

[Flanagan *et al.*, 1993] FLANAGAN, C., SABRY, A., DUBA, B. F. et FELLEISEN, M. (1993). The essence of compiling with continuations. *In PLDI*.

[Gehr *et al.*, 2016] GEHR, T., MISAILOVIC, S. et VECHEV, M. T. (2016). PSI : exact symbolic inference for probabilistic programs. *In CAV*.

[Gelman et Rubin, 1992] GELMAN, A. et RUBIN, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical science*, 7(4):457–472.

[Goodman et Stuhlmüller, 2014] GOODMAN, N. D. et STUHLMÜLLER, A. (2014). The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`.

[Hastings, 1970] HASTINGS, W. K. (1970). Monte Carlo sampling methods using markov chains and their applications. *Biometrika*, 57:97–109.

[Heunen *et al.*, 2017] HEUNEN, C., KAMMAR, O., STATON, S. et YANG, H. (2017). A convenient category for higher-order probability theory. *In LICS*.

[Hoffman et Gelman, 2014] HOFFMAN, M. D. et GELMAN, A. (2014). The No-U-Turn Sampler : adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623.

[Holtzen *et al.*, 2020] HOLTZEN, S., den BROECK, G. V. et MILLSTEIN, T. D. (2020). Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):140 :1–140 :31.

[Kozen, 1979] KOZEN, D. (1979). Semantics of probabilistic programs. *In SFCS*.

[Kucukelbir *et al.*, 2017] KUCUKELBIR, A., TRAN, D., RANGANATH, R., GELMAN, A. et BLEI, D. M. (2017). Automatic differentiation variational inference. *Journal of Machine Learning Research*, 18:14 :1–14 :45.

[Laplace, 1812] LAPLACE, P. (1812). *Théorie analytique des probabilités*. Courcier, Paris.

[Lee *et al.*, 2020] LEE, W., YU, H., RIVAL, X. et YANG, H. (2020). Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(POPL):16 :1–16 :33.

[Obermeyer et Bingham, 2020] OBERMEYER, F. et BINGHAM, E. (2020). Delayed sampling via barriers and funsors. *In PROBPROG*.

[Paszke *et al.*, 2019] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KÖPF, A., YANG, E. Z., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J. et CHINTALA, S. (2019). PyTorch : An imperative style, high-performance deep learning library. *In NeurIPS*.

[Salvatier *et al.*, 2016] SALVATIER, J., WIECKI, T. V. et FONNESBECK, C. (2016). Probabilistic programming in python using PyMC3. *PeerJ Computer Science*, 2:e55.

[Ścibior *et al.*, 2018] ŚCIBIOR, A., KAMMAR, O., VÁKÁR, M., STATON, S., YANG, H., CAI, Y., OSTERMANN, K., MOSS, S. K., HEUNEN, C. et GHAHRAMANI, Z. (2018). Denotational validation of higher-order bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(POPL):60 :1–60 :29.

[Stan Development Team, 2024] STAN DEVELOPMENT TEAM (2024). *Stan Reference Manual, version 2.34*. https://mc-stan.org/docs/reference-manual/.

[Staton, 2017] STATON, S. (2017). Commutative semantics for probabilistic programming. *In ESOP*.

[Tran *et al.*, 2017] TRAN, D., HOFFMAN, M. D., SAUROUS, R. A., BREVDO, E., MURPHY, K. et BLEI, D. M. (2017). Deep probabilistic programming. *In ICLR*.

[van de Meent *et al.*, 2018] van de MEENT, J., PAIGE, B., YANG, H. et WOOD, F. (2018). An introduction to probabilistic programming. *CoRR*, abs/1809.10756.

[Webb *et al.*, 2019] WEBB, S., CHEN, J. P., JANKOWIAK, M. et GOODMAN, N. D. (2019). Improving automated variational inference with normalizing flows. *In ICML Workshop on Automated Machine Learning*.

[Wingate *et al.*, 2011] WINGATE, D., STUHLMÜLLER, A. et GOODMAN, N. D. (2011). Lightweight implementations of probabilistic programming languages via transformational compilation. *In AISTATS*.