i- flow: High-dimensional integration and sampling with normalizing flows

IOP Publishing Mach. Learn.: Sci. Technol. 1 (2020) 045023 https://doi.org/10.1088/2632-2153/abab62

MACHINE LEARNING
Science and Technology

OPEN ACCESS

RECEIVED
18 February 2020

REVISED
24 July 2020

ACCEPTED FOR PUBLICATION
31 July 2020

PUBLISHED
12 November 2020

PAPER

i-flow: High-dimensional integration and sampling with normalizing flows

Christina Gao, Joshua Isaacson and Claudius Krause

Theoretical Physics Department, Fermi National Accelerator Laboratory, Batavia, IL, 60510, United States of America

E-mail: ckrause@fnal.gov

Keywords: normalizing flows, Monte Carlo integration, importance sampling, random number generators, Monte Carlo, density estimation

Abstract

In many fields of science, high-dimensional integration is required. Numerical methods have been developed to evaluate these complex integrals. We introduce the code `i-flow`, a Python package that performs high-dimensional numerical integration utilizing normalizing flows. Normalizing flows are machine-learned, bijective mappings between two distributions. `i-flow` can also be used to sample random points according to complicated distributions in high dimensions. We compare `i-flow` to other algorithms for high-dimensional numerical integration and show that `i-flow` outperforms them for high dimensional correlated integrals. The `i-flow` code is publicly available on gitlab at `https://gitlab.com/i-flow/i-flow`.

1. Introduction

Simulation based on first principles is an important practice, because it is the only way that a theoretical model can be checked against experiments or real-world data. In high-energy physics (HEP) experiments, a thorough understanding of the properties of known physics forms the basis of any searches that look for new effects. This can only be achieved by an accurate simulation, which in many cases boils down to performing an integral and sampling from it. Often high-dimensional phase space integrals with non-trivial correlations between dimensions are required in important theory calculations. Monte-Carlo (MC) methods still remain as the most important techniques for solving high-dimensional problems across many fields, including for instance: biology [1, 2], chemistry [3], astronomy [4], medical physics [5], finance [6] and image rendering [7]. In high-energy physics, all analyses at the Large Hadron Collider (LHC) rely strongly on multipurpose Monte Carlo event generators [8, 9] for signal or background prediction. However, the extraordinary performance of the experiments requires an amount of simulated data that soon cannot be delivered with current algorithms and computational resources [10, 11].

A main endeavour in the field of MC methods is to improve the error estimate. In particular, stratified sampling—dividing the integration domain in sub-domains, and importance sampling—sampling from non-uniform distributions [12] are two ways of reducing the variance. Currently, the most widely used numerical algorithm that exploits importance sampling is the `VEGAS` algorithm [13, 14]. But `VEGAS` assumes the factorizability of the integrand, which can be a bad approximation if the variables have complex correlations amongst one another. `Foam` [15] is a popular alternative that tries to address this issue. It uses an adaptive strategy to attempt to model correlations, but requires exponentially large samples in high dimensions.

Lately, the burgeoning field of machine learning (ML) has brought new techniques into the game. For the following discussion, we restrict ourselves to focus on progress made in the field of high-energy physics, see [16] for a recent review. However, these techniques are also widely applied in other areas of research. Concerning event generation, [17] used boosted decision trees and generative adversarial networks (GANs) to improve MC integration. Reference [18] proposed a novel idea that uses a dense neural network (DNN) to

learn the phase space directly and shows promising results. In principle, once the neural network (NN)-based algorithm for MC integration is trained, one can invert the network and use it for sampling. However, the inversion of the NN requires evaluating its Jacobian, which incurs a computational cost that scales as $\mathcal{O}(D^3)$ for D-dimensional integrals [1]. Therefore, it is extremely inefficient to use a standard NN-based algorithm for sampling.

In addition to generating events from scratch, it is possible to generate additional events from a set of precomputed events. References [19–25] used GANs and Variational Autoencoders (VAEs) to achieve this goal. While their work is promising, they have a few downsides. The major advantage of this approach is the drastic speed improvement over standard techniques. They report improvements in generation of a factor around 1000. However, this approach requires a significant number of events already generated which may be cost prohibitive for interesting, high-multiplicity problems. Furthermore, these approaches can only generate events similar to those already generated. Therefore, this would not improve the corners of distributions [26] and can even result in incorrect total cross-sections. Yet another approach to speed up event generation is to use NN as interpolator and learn the Matrix Element [27].

Our goal is to explore NN architectures that allow both efficient MC integration and sampling. A ML algorithm based on *normalizing flows* (NF) provides such a candidate. The idea was first proposed by *non-linear independent components estimation* (NICE) [28, 29], and generalized in [30–32], for example. They introduced *coupling layers* (CL) allowing the inclusion of NNs in the construction of a bijective mapping between the target and initial distributions such that the $\mathcal{O}(D^3)$ evaluation of the Jacobian can be reduced to an analytic expression. This expression can now be evaluated in $\mathcal{O}(D)$ time. These techniques have also been combined with Markov Chain Monte Carlo methods, showing promising results [33–35].

Our contribution is a complete, openly available implementation of normalizing flows into `TensorFlow` [36], to be used for any high-dimensional integration problem at hand. Our code includes the original proposal of [31] and the additions of [32]. We further include various different loss functions, based on the class of f-divergences [37]. The paper is organized in the following way. The basic principles of MC integration and importance sampling are reviewed in section 2. In section 3, we review the concept of normalizing flows and work done on CL-based flow by [28, 29, 31, 32]. We investigate the minimum number of CLs required to capture the correlations between every other input dimension. Section 4 sets up the stage for a comparison between our code, `VEGAS`, and `Foam` on various trial functions, of which we give results in section 5. This comparison is based on several criteria, allowing a potential user to judge whether it might be worth trying out. Section 6 contains our conclusion and outlook.

2. Monte Carlo integrators

While techniques exist for accurate one-dimensional integration, such as double exponential integration [38], using them for high dimensional integrals requires repeated evaluation of one dimensional integrals. This leads to an exponential growth in computation time as a function of the number of dimensions. This is often referred to as the *curse of dimensionality*. In other words, when the dimensionality of the integration domain increases, the points become more and more sparse and no statistically significant statement can be made without increasing the number of points exponentially. This can be seen in the ratio of the volume of a D-dimensional hypersphere to the D-dimensional hypercube, which vanishes as D goes to infinity. However, Monte-Carlo techniques are statistical in nature and thus always converge as $1/\sqrt{N}$ for any number of dimensions.

Therefore, MC integration is the most important technique in solving high-dimensional integrals numerically. The naive MC approach samples uniformly on the integration domain (Ω). Given N uniform samples, the integral of $f(x)$ can be approximated by,

$$I \approx \frac{V}{N}\sum_{i=1}^{N} f(x_i) \equiv V\langle f\rangle_x, \tag{1}$$

and the uncertainty is determined by the standard deviation of the mean,

$$\sigma_I = \sqrt{\mathrm{Var}} \approx V\sqrt{\frac{\langle f^2\rangle_x - \langle f\rangle_x^2}{N-1}}\,, \tag{2}$$

[1] An N particle final state phase space is a $D \approx 4N-3$ dimensional integral, when including recursive multichannel selection in the integral.

where V is the volume encompassed by Ω and $\langle\,\rangle_x$ indicates that the average is taken with respect to a uniform distribution in x. While this works for simple or low-dimensional problems, it soon becomes inefficient for high-dimensional problems. This is what our work is concerned with. In particular, we are going to focus on improving current methods for MC integration that are based on importance sampling.

In importance sampling, instead of sampling from an uniform distribution, one samples from a distribution $g(x)$ that ideally has the same shape as the integrand $f(x)$. Using the transformation $\mathrm{d}x = \mathrm{d}G(x)/g(x)$, with $G(x)$ the cumulative distribution function of $g(x)$, one obtains

$$I = \int_\Omega \frac{f(x)}{g(x)}\mathrm{d}G(x) = V\langle f/g\rangle_G\,, \qquad \sigma_I = V\sqrt{\frac{\langle (f/g)^2\rangle_G - \langle f/g\rangle_G^2}{N-1}}\,. \tag{3}$$

In the ideal case when $g(x)\to f(x)/I$, equation 3 would be estimated with vanishing uncertainty. However, this requires already knowing the analytic solution to the integral! The goal is thus to find a distribution $g(x)$ that resembles the shape of $f(x)$ most closely, while being integrable and invertible such as to allow for fast sampling. We review the current MC integrators that are widely used, especially in the field of high-energy physics.

`VEGAS` [13, 14] approximates all 1-dimensional projections of the integrand using a histogram and an adaptive algorithm. This algorithm adjusts the bin widths such that the area of the bins are roughly equal. To sample a random point from `VEGAS` can be done in two steps. First, select a bin randomly for each dimension. Second, sample a point from each bin according to a uniform distribution. However, this algorithm is limited because it assumes that the integrand factorizes, i.e.

$$f(\vec{x}) = f_1(x_1)\cdots f_D(x_D)\,, \tag{4}$$

where $f\colon \mathbb{R}^D \mapsto \mathbb{R}$ and $f_i\colon \mathbb{R}\mapsto\mathbb{R}$. High-dimensional integrals with non-trivial correlations between integration variables, that are often needed for LHC data analyses, cannot be integrated efficiently with the `VEGAS` algorithm (c.f. [39]). The resulting uncertainty can be reduced further by applying stratified sampling, in addition to the `VEGAS` algorithm, after the binning [40].

`Foam` [15] uses a cellular approximation of the integrand and is therefore able to learn correlations between the variables. In the first phase of the algorithm, the so-called exploration phase, the cell grid is built by subsequent binary splits of existing cells. Since the first cell consists of the full integration domain, all regions of the integration space are explored by construction. The second phase of the algorithm uses this grid to generate points either for importance sampling or as an event generator. In this work we use the implementation of [41], which implemented an additional reweighting of the cells at the end of the optimization.

However, both `Foam` and `VEGAS` are based on histograms, whose edge effects would be detrimental to numerical analyses that demand high precision. As we will explain below, our code `i-flow` uses a spline approximation which does not suffer from these effects. These edge effects are an important source of uncertainty for high-precision physics [42].

3. Importance sampling with normalizing flows

As we detailed in the previous section, importance sampling requires finding an approximation $g(x)$ that can easily be integrated and subsequently inverted, so that we can use it for sampling. Mathematically, this corresponds to a coordinate transformation with an inverse Jacobian determinant that is given by $g(x)$. General ML algorithms incorporate NNs in learning the transformation, which inevitably involve evaluating the Jacobian of the NNs. This results in inefficient sampling. Coupling Layer-based Normailzing Flow algorithms precisely circumvent this problem. To begin, let us review the concept of a normalizing flow (NF).

Let c_k, with $k = 1, ..., K$, be a series of bijective mappings on the random variable $\vec{x}$:

$$\vec{x}_K = c_K(c_{K-1}(\cdots c_2(c_1(\vec{x})))). \tag{5}$$

Based on the chain rule, the output $\vec{x}_K$'s probability distribution, g_K, can be inferred given the base probability distribution g_0 from which $\vec{x}$ is drawn:

$$g_K(\vec{x}_K) = g_0(\vec{x}_0)\prod_{k=1}^{K}\left|\frac{\partial c_k(\vec{x}_{k-1})}{\partial \vec{x}_{k-1}}\right|^{-1}, \quad \text{where} \quad \begin{cases}\vec{x}_0 = \vec{x}\\ \vec{x}_k = c_k(\vec{x}_{k-1})\end{cases}. \tag{6}$$

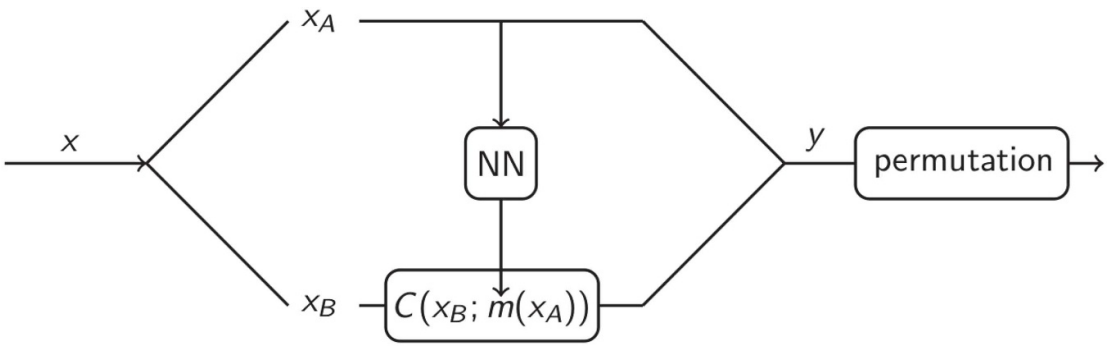


Figure 1. Structure of a Coupling Layer. m is the output of a neural network and defines the Coupling Transform, C, that will be applied to x_B. See equations (7), (8), (9) for the mathematical description of a Coupling Layer.

One sees that the target and base distributions are related by the inverse Jacobian determinant of the transformation. For practical uses, the Jacobian determinant must be easy to compute, restricting the allowed functional forms of c_k. However, with the help of *coupling layers*, first proposed by [28, 29], then generalized by [31, 32], one can incorporate NNs into the construction of c_k, thus greatly enhancing the level of complexity and expressiveness of NF without introducing any intractable Jacobian computations.

Figure 1 shows the basic structure of a coupling layer, which is a special design of the bijective mapping c. For each map, the input variable $\vec{x} = \{x_1, .., x_D\}$ is partitioned into two subsets, $\vec{x}_A$ and $\vec{x}_B$, which can be determined arbitrarily so long as neither is the empty set. This arbitrary partitioning will be referred to as a *masking*. Without loss of generality, one simple partitioning is given by $\vec{x}_A = \{x_1, .., x_d\}$ and $\vec{x}_B = \{x_{d+1}, .., x_D\}$. Different maskings can be achieved via permutations of the simple example above. Under the bijective map, C, the resulting variable transforms as

$$\begin{aligned} x_A &= x_A, & A &\in [1, d] \,, \\ x_B &= C(x_B; m(\vec{x}_A)), & B &\in [d+1, D] \,. \end{aligned} \tag{7}$$

The NN takes x_A as inputs and outputs $m(\vec{x}_A)$ that represents the parameters of the invertible 'Coupling Transform', C, that will be applied to x_B. We detail various choices for C, like piecewise linear, piecewise quadratic, or piecewise rational quadratic spline functions in appendix A. The inverse map is given by

$$\begin{aligned} x_A &= x_A \,, \\ x_B &= C^{-1}(x_B; m(\vec{x}_A)) = C^{-1}(x_B; m(\vec{x}_A)) \,, \end{aligned} \tag{8}$$

which leads to the simple Jacobian

$$\left| \frac{\partial c(\vec{x})}{\partial \vec{x}} \right|^{-1} = \left| \begin{pmatrix} \vec{1} & 0 \\ \frac{\partial C}{\partial m} \frac{\partial m}{\partial \vec{x}_A} & \frac{\partial C}{\partial \vec{x}_B} \end{pmatrix} \right|^{-1} = \left| \frac{\partial C(\vec{x}_B; m(\vec{x}_A))}{\partial \vec{x}_B} \right|^{-1} . \tag{9}$$

Note that equation (9) does not require the computation of the gradient of $m(\vec{x}_A)$, which would scale as $\mathcal{O}(D^3)$ with D the number of dimensions. In addition, taking $\partial C / \partial \vec{x}_B$ to be diagonal further reduces the computation complexity of the determinant to be linear with respect to the dimensionality of the problem. Linear scaling makes this approach tractable even for high dimensional problems. In summary, the NN learns the parameters of a transformation and not the transformation itself, thus the Jacobian can be calculated analytically.

To construct a complete Normalizing Flow, one simply compounds a series of Coupling Layers with the freedom of choosing any of the outputs of the previous layer to be transformed in the subsequent layer. We show in Section 3.1 that $2\lceil \log_2 \mathrm{D} \rceil$ number of Coupling Layers are required in order to express all non-separable structures of the integrand.

3.1. Number of coupling layers

The minimum number of coupling layers required to capture all possible correlations between every dimension of the integration variable, $n_{\min}$, depends on the dimensionality of the integral, D [31]. In the cases of $D = 2$ and $D = 3$, each dimension is transformed once based on the other dimension(s) and thus $n_{\min} = 2$ and $n_{\min} = 3$, respectively. This way of counting $n_{\min}$ could be generalized to higher D. In fact, this

is what *autoregressive flows* are based on [43]. Here we show that the number of coupling layers required to capture all the correlations is $2\lceil \log_2 D \rceil$ for $D > 5$, and D layers for $D \leq 5$. This can be considered the minimum number of layers required in order to capture all correlations, adding an additional layer will not add any new information, and similar effects should be achieved with increasing the depth of the network associated with each layer. On the other hand, this can be considered the maximum number of layers needed to capture all the correlations. If a function has fewer correlations, then all the correlations can be captured with less than $2\lceil \log_2 D \rceil$.

Theorem. *Given a set of correlated random variables x, if a transformation exists that takes the variables x to z, such that the correlation between the variables z is zero, then a composition of normalizing flows can create such a transformation. Given a set of infinitely wide NNs that are universal function approximators, and requiring that all variables are transformed equal number of times, it is possible to represent all the correlations between variables in a normalizing flow using $2\lceil \log_2 D \rceil$ layers for $D > 5$. When $D \leq 5$ it is possible to represent all correlations with D layers.*

Proof. Given the random variables $x_1, \ldots x_D$, with means $\mu_1, \ldots, \mu_D$ and joint probability distribution $f(x_1, \ldots, x_D)$, the correlation between all the variables is given by:

$$\langle (x_1 - \mu_1) \ldots (x_D - \mu_D) \rangle = \int_0^1 dx_1 \ldots dx_D \, (x_1 - \mu_1) \ldots (x_D - \mu_D) f(x_1, \ldots, x_D) \,. \tag{10}$$

Using two layers of a normalizing flow network, which can be seen as a universal function approximator, defines a transformation $T : x \mapsto y$, with the bounds of integration being mapped such that $y_i(T(x=0)) = 0$ and $y_i(T(x=1)) = 1 \; \forall i \in [1, D]$, with the sets $\{y_a\} = \{y_i | \, i \equiv 1 \pmod 2, i \in [1, D]\}$ and $\{y_b\} = \{y_i | \, i \equiv 0 \pmod 2, i \in [1, D]\}$, such that $f(x_1, \ldots x_D) \mapsto g(\{y_a\}) \, h(\{y_b\})$, and with the Jacobian $J(y, x)$. The transformation also maps the means: $\mu \mapsto \mu^y$. This decomposition is possible following from the arguments section 2.2 of [44]. Applying the transformation to equation (10) gives:

$$\langle (x_1 - \mu_1) \ldots (x_D - \mu_D) \rangle = \int_0^1 dy_1 \ldots dy_D J(y, x) \left(y_1 - \mu_1^y\right) \ldots \left(y_D - \mu_D^y\right) g(\{y_a\}) \, h(\{y_b\}) \,. \tag{11}$$

If we now consider the correlation between the variables y, we obtain:

$$\begin{aligned} \left\langle \left(y_1 - \mu_1^y\right) \ldots \left(y_D - \mu_D^y\right) \right\rangle &= \int_0^1 dy_1 \ldots dy_D \left(y_1 - \mu_1^y\right) \ldots \left(y_D - \mu_D^y\right) g(\{y_a\}) \, h(\{y_b\}) \\ &= \int_0^1 \prod_{\{y_a\}} \left(dy_a \left(y_a - \mu_a^y\right)\right) g(\{y_a\}) \times \int_0^1 \prod_{\{y_b\}} \left(dy_b \left(y_b - \mu_b^y\right)\right) h(\{y_b\}) \\ &= \left\langle \prod_{\{y_a\}} \left(y_a - \mu_a^y\right) \right\rangle \left\langle \prod_{\{y_b\}} \left(y_b - \mu_b^y\right) \right\rangle . \end{aligned} \tag{12}$$

The result of the transformation shows that the variables y_a are now not correlated with y_b. We can construct a subsequent transformation $T : y \mapsto z$, with $z_i\left(T^{(y=0)}\right) = 0$ and $z_i\left(T^{(y=1)}\right) = 1 \; \forall i \in [1, D]$, and the sets $\{z_a\} = \{z_i | \, i \equiv 1 \pmod 4, i \in [1, D]\}$, $\{z_b\} = \{z_i | \, i \equiv 2 \pmod 4, i \in [1, D]\}$, $\{z_c\} = \{z_i | \, i \equiv 3 \pmod 4, i \in [1, D]\}$, and $\{z_d\} = \{z_i | \, i \equiv 0 \pmod 4, i \in [1, D]\}$, such that $g(\{y_a\}) \, h(\{y_b\}) \mapsto g^{(\{z_a, z_b\}) h^{(\{z_c, z_d\})}}$ with the constraint that such a transformation does not introduce new correlations between the variables that have already been decorrelated. In other words, the composition of T and T can be defined as a transformation T: $x \mapsto z$, with $z_i(T^{(x)} = 0) = 0$ and $z_i(T^{(x)} = 1) = 1 \; \forall i \in [1, D]$, such that:

$$f(x_1, \ldots, x_N) \mapsto g_1(\{z_a\}) \, g_2(\{z_b\}) \, g_3(\{z_c\}) \, g_4(\{z_d\}) \,,$$

and the means are mapped from μ to μ^z. Thus, the correlation between the variables z is given by:

$$\begin{aligned} \langle (z_1 - \mu_1^z) \ldots (z_D - \mu_D^z) \rangle &= \int_0^1 dz_1 \ldots dz_D \, (z_1 - \mu_1^z) \ldots (z_D - \mu_D^z) \, g_1(\{z_a\}) \, g_2(\{z_b\}) \, g_3(\{z_c\}) \, g_4(\{z_d\}) \\ &= \left\langle \prod_{\{z_a\}} (z_a - \mu_a^z) \right\rangle \left\langle \prod_{\{z_b\}} (z_b - \mu_b^z) \right\rangle \left\langle \prod_{\{z_c\}} (z_c - \mu_c^z) \right\rangle \left\langle \prod_{\{z_d\}} (z_d - \mu_d^z) \right\rangle . \end{aligned} \tag{13}$$

The above transformations can be iterated until all the variables are decorrelated. A method of determining the mapping for each step can be obtained by the following procedure:

Table 1. Finding the unique masking to capture all correlations in a $D = 12$ space, using the procedure detailed above.

Dimension	0	1	2	3	4	5	6	7	8	9	10	11
Transformation 1	0	1	0	1	0	1	0	1	0	1	0	1
Transformation 2	0	0	1	1	0	0	1	1	0	0	1	1
Transformation 3	0	0	0	0	1	1	1	1	0	0	0	0
Transformation 4	0	0	0	0	0	0	0	0	1	1	1	1

(a) Reindex the dimension numbers from $[1, D]$ to $[0, D-1]$
(b) Convert all dimensions to their binary representation, using the minimum number of bits required to represent the number $D-1$
(c) Consider the least significant bit for each dimension, and define the transformation as $T : x \mapsto y$, with $f(\{x_0\}, \{x_1\}) \mapsto g(\{x_0\})h(\{x_1\})$, where $\{x_0\}(\{x_1\})$ is the set of variables with a 0 (1) for the least significant bit
(d) Repeat the third step taking the next least significant bit, until the most significant bit is reached

See table 1 for an example of the steps above. In that example, transformation 1 would groups the first 8 dimensions in $g(x_0)$ and the last 4 in $h(x_1)$ etc

The number of steps for this procedure can easily be seen to be $\lceil \log_2(D) \rceil$. However, since we need two layers per transformation to ensure that all variables are equally transformed by the network leads to a requirement of $2\lceil \log_2(D) \rceil$.

In the situation of $D \leq 5$, the ith coupling layer can be defined to take the ith variable and transform the variable, such that it is not correlated with any other variable. This leads to a requirement of D layers.

The requirement on the above theorem is that we require that a transformation exists in order to perform the above mapping. However, even if a transformation does not exist for the integrand itself, with the use of importance sampling, it is only necessary to find a function g which is as close to the integrand as possible. In `i-flow` splines are used to create the function g, and according to the Stone-Weierstrass Theorem [45–47], it is possible to represent g such that it is ε close to f. A corollary of the Stone-Weierstrass Theorem for $\mathbb{R}^n$ can be expressed as:

Corollary. *Given a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, $\varepsilon > 0$, and $C(\mathbb{R}^n, \mathbb{R})$: the space of all real-valued continuous functions in $\mathbb{R}^n$, there exists a polynomial spline $g \in C(\mathbb{R}^n, \mathbb{R})$ such that:*

$$|f(x) - g(x)| < \varepsilon, \tag{14}$$

for all $x \in \mathbb{R}^n$.

Proof. The space R^n is a subset of the spaces proved in the Stone-Weierstrass Theorem, and thus the proof of the corollary follows directly from the Stone-Weierstrass Theorem.

Furthermore, this can be extended to a sum of piecewise polynomials, such that any continuous and bounded function f can be represented by an infinite series of polynomials (see Theorem C from [45]). In `i-flow`, we will consider the case of discontinuous functions, but these can be approximated as a continuous function with a slope of $1/\varepsilon$ in the region of discontinuity. This will lead to some difference between f and g, but since the goal is to find a function g as close to f as possible, then this is acceptable and should still allow for high precision importance sampling.

3.2. Using `i-flow`

The `i-flow` package requires three pieces of information from the user: the function to be integrated, the normalizing flow network, and the method of optimizing the network. Figure 2 shows schematically how one step in the training of `i-flow` works. The code is publicly available on gitlab at `https://gitlab.com/i-flow/i-flow`. Running the script `iflow_test.py` will produce the results presented in section 5.

3.2.1. Integrand

The function to be integrated has very few requirements on how it is implemented in the code. Firstly, the function must accept an array of points with shape (n_{batch}, D), where n_{batch} is the number of points to sample per training step. Secondly, the function must return an array with shape (n_{batch}) to be used to estimate the integral. Finally, the number of dimensions in the integral is required to be at least 2. However, one

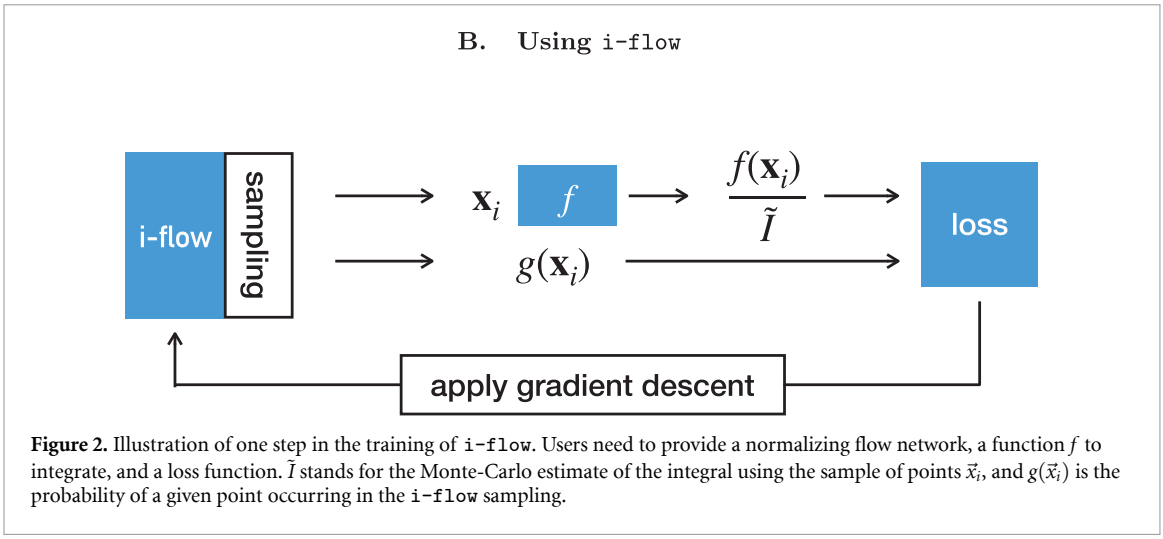


Figure 2. Illustration of one step in the training of `i-flow`. Users need to provide a normalizing flow network, a function f to integrate, and a loss function. $\tilde{I}$ stands for the Monte-Carlo estimate of the integral using the sample of points $\vec{x}_i$, and $g(\vec{x}_i)$ is the probability of a given point occurring in the `i-flow` sampling.

dimension can be treated as a dummy dimension integrated from 0 to 1, which will not change any result.

3.2.2. Normalizing flow network

A normalizing flow network consists of a series of coupling layers compounded together. To construct each coupling layer, one needs to specify the choice of coupling transform C (cf appendix. A), the number of coupling layers, the masking for each level, and the neural network $m(x_A)$ that constitutes the coupling transform. We provide the ability to automatically generate the masking and number of layers according to section 3.1.

The neural networks $m(x_A)$ must be provided by the user. However, we provide examples for a dense network and the U-shape network of [31]. This provides the user the flexibility to achieve the expressiveness required for their specific problem.

3.2.3. Optimizing the network

To uniquely define the optimization algorithm of the network, two pieces of information are required. Firstly, the loss function to be minimized is required. We supply a large set of loss functions from the set of f-divergences, which can be found in appendix B. By default, the `i-flow` code uses the exponential loss function. Secondly, an optimizer needs to be supplied. In the examples we used the ADAM optimizer [48]. However, the code can use any optimizer implemented within `TensorFlow`.

3.2.4. Hyperparameters

The setup we presented here has several hyperparameters that can be adjusted for better performance. However, `i-flow` has the flexibility for the user to implement additional features in each section beyond what is discussed below. This would come with additional hyperparameters as well.

The first group concerns the architecture of the NNs $m(x_A)$. Once the general type of network (dense or U-shape) is set, the number of layers and nodes per layer have to be specified. In the case of the U-shape network, the user can specify the number of nodes in the first layer and the number of 'downward' steps.

The second group of hyperparameters concerns the optimization process. Apart from setting an optimizer (*e.g.* ADAM [48]), a learning schedule (*e.g.* constant or exponentially decaying), an initial learning rate, and a loss function have to be specified. Some of these options come with their own, additional set of hyperparameters. The number of points per training epoch and the number of epochs have to be set as well.

The third group of hyperparameters concerns the setup of `i-flow` directly. As was discussed in [31], there are two ways to pass x_A into $m(x_A)$: either directly or with one-blob encoding. `i-flow` supports both of these options. One-blob encoding [31] is a generalization of one-hot encoding. The input x_A is passed through a Gaussian kernel and several adjacent bins are activated. If one-blob encoding is used, the number of input bins has to be specified, the width of the Gaussian is set to the inverse of the number of bins. Further, the type of coupling function $C(x_B, m(x_A))$, the number of output bins, the number of CLs and the maskings have to be set.

3.2.5. Putting it all together

The networks are trained by sampling a fixed number of points using the current state of $g(x)$[2]. We use one of the statistical divergences as a measure for how much the distribution $g(x)$ resembles the shape of the integrand $f(x)$, and an optimizer to minimize it. Because we can generate an infinite set of random numbers and evaluate the target function for each of the points, this approach corresponds to supervised learning with an infinite dataset. Drawing a new set of points at every training epoch automatically also ensures that the networks cannot overfit.

4. Integrator Comparison

To illustrate the performance of `i-flow` and compare it to `VEGAS` and `Foam`, we present a set of six test functions, each highlighting a different aspect of high-dimensional integration and sampling. These functions demonstrate how each algorithm handles the cases of a purely separable function, functions with correlations, and functions with non-factorizing hard cuts. In most cases, an analytic solution to the integral is known.

The first test function is an n-dimensional Gaussian, serving as a sanity check:

$$f_1(\vec{x}) = (\alpha\sqrt{\pi})^{-n} \exp\{-\frac{\sum_i (x_i - 0.5)^2}{\alpha^2}\} . \tag{15}$$

The result of integrating f_1 from zero to one is given by:

$$\int_0^1 d^n\vec{x} f_1(\vec{x}) = \text{erf}\left(\frac{1}{2\alpha}\right)^n . \tag{16}$$

In the following, we use $\alpha = 0.2$.

The second test function is an n-dimensional Camel function, which would show how `i-flow` learns correlations that `VEGAS` (without stratified sampling) would not learn:

$$f_2(\vec{x}) = \frac{1}{2}(\alpha\sqrt{\pi})^{-n}\left(\exp\{-\frac{\sum_i (x_i - \frac{1}{3})^2}{\alpha^2}\} + \exp\{-\frac{\sum_i (x_i - \frac{2}{3})^2}{\alpha^2}\}\right) . \tag{17}$$

The result of integrating f_2 from zero to one is given by:

$$\int_0^1 d^n\vec{x} f_2(\vec{x}) = \left(\frac{1}{2}\left(\text{erf}\left(\frac{1}{3\alpha}\right) + \text{erf}\left(\frac{2}{3\alpha}\right)\right)\right)^n . \tag{18}$$

In the following, we use $\alpha = 0.2$.

The third case is given by

$$\begin{aligned} f_3(x_1, x_2) = {} & x_2^a \exp\{-w|(x_2 - p_2)^2 + (x_1 - p_1)^2 - r^2|\} \\ & + (1 - x_2)^a \exp\{-w|(x_2 - 1 + p_2)^2 + (x_1 - 1 + p_1)^2 - r^2|\} . \end{aligned} \tag{19}$$

This function has two circles with shifted centers, varying thickness and height. Also, the function exhibits non-factorizing behavior. The integral of f_3 between 0 and 1 can be computed numerically using Mathematica [49], which is $0.013\,684\,8 \pm (5 \cdot 10^{-9})$, with $p_1 = 0.4, p_2 = 0.6, r = 0.25, w = 1/0.004$ and $a = 3$[3].

The fourth case is an annulus function with hard cuts:

$$f_4(x_1, x_2) = \begin{cases} 1 & 0.2 < \sqrt{x_1^2 + x_2^2} < 0.45 \\ 0 & \text{else} \end{cases} . \tag{20}$$

This function demonstrates how `i-flow` learns hard, non-factorizing cuts. The result of integrating f_4 from zero to one is given by: $\pi\left(0.45^2 - 0.2^2\right) = 0.162\,5\pi$.

The fifth case is motivated by high energy physics, and is a one-loop scalar box integral representative of an integral required for the calculation of $gg \to gh$ in the Standard Model. This calculation is an important contribution for the total production cross-section of the Higgs boson. As explained in appendix C, after Feynman parametrisation and sector decomposition [50], the integral of interest is given by

[2] Since we initialize the last layer of each network with vanishing bias and weights, in the first sampling $g(x)$ is constant.

[3] There is no known analytic solution to this given function.

Table 2. Number of functional calls to reach a total relative uncertainty of 10^{-4} (for the first 11 cases) or 10^{-5} (for the last 3 cases). The total relative uncertainty is defined as the inverse-variance weighted combination of the uncertainties of each optimization iteration divided by the true integral value. The integrator with the fewest functional calls, which also is within 5 standard deviations of the true result, is highlighted in boldface. We set an upper cut-off of $5 \cdot 10^7$ calls. A † indicates that the algorithm did not converge to the true integral value within 5 standard deviations (see table 3), a * indicates cases where the algorithm ran out of memory before the cut-off was reached. $\alpha = 0.2$ for Gaussian and Camel functions.

	Dim	VEGAS	Foam	i-flow
Gaussian	2	**164,436**	6, 259, 812	2, 310, 000
	4	**631,874**	24, 094, 679	2, 285, 000
	8	**1,299,718**	> 50, 000, 000†	3, 095, 000
	16	**2,772,216**	> 50, 000, 000†	7, 230, 000
Camel	2	**421,475**	5, 619, 646	2, 225, 000
	4	24, 139, 889	21, 821, 075	**8, 220, 000**
	8	> 50, 000, 000†	> 50, 000, 000†	**19, 460, 000**
	16	993, 294 †	> 50, 000, 000†	32, 145, 000 †[5]
Entangled circles	2	43, 367, 192	**17,499,823**	23, 105, 000
Annulus w. cuts	2	4, 981, 080 †	**11,219,498**	17, 435, 000
Scalar-top-loop	3	**152, 957**	5, 290, 142	685, 000
Polynomial	18	42, 756, 678	> 50, 000, 000	**585, 000**
	54	> 50, 000, 000	> 21, 505, 000 *	**685, 000**
	96	> 50, 000, 000†	> 10, 325, 000*	> **1, 145, 000**

$$
\begin{aligned}
f_5 =& S_{Box}(s_{12}, s_{23}, s_1, s_2, s_3, s_4, m_t^2, m_t^2, m_t^2, m_t^2) \\
&+ S_{Box}(s_{23}, s_{12}, s_2, s_3, s_4, s_1, m_t^2, m_t^2, m_t^2, m_t^2) \\
&+ S_{Box}(s_{12}, s_{23}, s_3, s_4, s_1, s_2, m_t^2, m_t^2, m_t^2, m_t^2) \\
&+ S_{Box}(s_{23}, s_{12}, s_4, s_1, s_2, s_3, m_t^2, m_t^2, m_t^2, m_t^2) \\
S_{Box}(s_{12}, s_{23}, s_1, s_2, s_3, s_4, m_1^2, m_2^2, m_3^2, m_4^2) =& \int_0^1 dt_1 dt_2 dt_3 \frac{1}{\tilde{\mathcal{F}}_{Box}^2} \\
\tilde{\mathcal{F}}_{Box} =& (-s_{12})t_2 + (-s_{23})t_1 t_3 + (-s_1)t_1 + (-s_2)t_1 t_2 + (-s_3)t_2 t_3 \\
&+ (-s_4)t_3 + (1 + t_1 + t_2 + t_3)(t_1 m_1^2 + t_2 m_2^2 + t_3 m_3^2 + m_4^2)
\end{aligned}
\tag{21}
$$

The result of integrating f_5 from zero to one can be obtained through the use of LoopTools [51], which gives a numerical result of $1.936\,964\,023\,8 \cdot 10^{-10}$ for the inputs $s_{12} = 130^2, s_{23} = -130^2, s_1 = 0, s_2 = 0, s_3 = 0, s_4 = 125^2, m_t = 175$.

As a sixth test function, we consider the polynomial

$$
f_6(x_1, \ldots, x_n) = \sum_{i=1}^{n} -x_i^2 + x_i. \tag{22}
$$

The result of integrating f_6 from zero to one is given by:

$$
\int_0^1 dx_1 \ldots dx_n f_6(x_1, \ldots, x_n) = \frac{n}{6} \tag{23}
$$

This function can easily be integrated in a high number of dimensions and, unlike the Gaussian or Camel functions, has support in almost all of the integration domain. It therefore does not suffer that much from the curse of dimensionality.

Further applications to event generation of high-energy particle collisions is discussed in [52] and also in [53]. These papers investigate using normalizing flows to improve upon phase space integration for event simulation at particle colliders. The integral dimension for processes with n_f particles in the final state is $D = 4n_f - 3$. In [52], we studied processes with $n_f \leq 6$.

5. Results

In this section we show the performance of i-flow and compare it to VEGAS and Foam based on the test functions we introduced in section 4. For the VEGAS algorithm, we use the default parameters as implemented in [40]. This includes the use of stratified sampling and a maximum of 1000 bins per axis. We further set the number of points per iteration to 5000. However, the implementation in [40] uses this number as a maximum, so we monitor the actual number of function calls separately. The setup of Foam

Table 3. Integral estimate and uncertainty of the runs of table 2 together with their relative deviations ('pull'), defined in equation 25. A † indicates that the algorithm reached a cut-off of $5 \cdot 10^7$ function calls before the target uncertainty was reached, a * indicates cases where the algorithm ran out of memory before the cut-off was reached. The result with the smallest relative deviation is boldfaced. $\alpha = 0.2$ for Gaussian and Camel functions.

	Dim	VEGAS	(pull)	Foam	(pull)	i-flow	(pull)	true value
Gaussian	2	0.999 25(10)	0.7	0.999 25(10)	0.6	**0.999 19(10)**	**0.1**	0.999 186
	4	0.998 61(10)	2.4	**0.998 35(10)**	**−0.2**	0.998 41(10)	0.4	0.998 373
	8	0.996 94(10)	1.9	0.994 39(37) †	−6.4	**0.996 84(10)**	**0.9**	0.996 749
	16	0.993 57(10)	0.6	0.549 86(235) †	−188	**0.993 54(10)**	**0.4**	0.993 509
Camel	2	0.981 75(10)	0.9	0.981 63(10)	−0.3	**0.981 65(10)**	**−0.1**	0.981 66
	4	0.963 45(10)	−2.2	0.963 61(10)	−0.5	**0.963 65(10)**	**−0.02**	0.963 657
	8	0.924 95(28) †	−13	0.927 98(19) †	−3.5	**0.928 43(9)**	**−2.2**	0.928 635
	16	0.431 37(9)	−5001	0.769 21(129) †	−72	**0.859 40(9)** [55]	**−34**	0.862 363
Entangled circles	2	0.013 679 8(14)	−3.6	**0.013 683 8(14)**	**−0.7**	0.013 682 9(14)	−1.4	0.013 684 8
Annulus w. cuts	2	0.509 813(51)	−14	0.510 559(51)	1.0	**0.510 511(51)**	**0.1**	0.510 508
Scalar-top-loop	3	$1.937\,11(19) \cdot 10^{-10}$	0.7	$\mathbf{1.937\,08(19) \cdot 10^{-10}}$	**0.6**	$1.936\,77(19) \cdot 10^{-10}$	−1.0	$1.936\,964 \cdot 10^{-10}$
Polynomial	18	2.999 89(3)	−3.6	2.999 86(12) †	−1.1	**2.999 97(3)**	**−1.1**	3
	54	8.999 72(19) †	−1.5	9.000 13(32) *	0.4	**9.000 01(9)**	**0.2**	9
	96	0.155 47(52) †	−30683	16.000 4(3) *	1.7	**15.999 8(2)**	**−1.2**	16

Table 4. Relative uncertainty on the integral estimate of the last iteration of the runs of table 2, based on a sample of 5000 points. The integrator that adapted best to the integrand is boldfaced. A * indicates when the value was still decreasing and had not yet converged, a † is in place where the algorithm did not converge to the true integrand.

	Dim	VEGAS	Foam	i-flow
Gaussian	2	$\mathbf{7 \cdot 10^{-4}}$	$3 \cdot 10^{-3}$	$2 \cdot 10^{-3}$ *
	4	$\mathbf{1.5 \cdot 10^{-3}}$	$3 \cdot 10^{-3}$	$\mathbf{1.5 \cdot 10^{-3}}$ *
	8	$2.5 \cdot 10^{-3}$	$3 \cdot 10^{-2}$	$\mathbf{1.5 \cdot 10^{-3}}$ *
	16	$3.5 \cdot 10^{-3}$	$2 \cdot 10^{-2}$	$\mathbf{2.5 \cdot 10^{-3}}$ *
Camel	2	$\mathbf{2 \cdot 10^{-3}}$	$\mathbf{2 \cdot 10^{-3}}$	$\mathbf{2 \cdot 10^{-3}}$ *
	4	$8 \cdot 10^{-3}$	$1 \cdot 10^{-2}$	$\mathbf{4 \cdot 10^{-3}}$
	8	$4 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$	$\mathbf{5 \cdot 10^{-3}}$
	16	†	$1.5 \cdot 10^{-1}$	$\mathbf{5 \cdot 10^{-3}}$
Entangled circles	2	$1 \cdot 10^{-2}$	$\mathbf{4 \cdot 10^{-3}}$	$5 \cdot 10^{-3}$ *
Annulus w. cuts	2	$\mathbf{3 \cdot 10^{-3}}$	$4 \cdot 10^{-3}$ *	$5 \cdot 10^{-3}$
Scalar-top-loop	3	$7 \cdot 10^{-4}$	$\mathbf{5 \cdot 10^{-4}}$	$\mathbf{5 \cdot 10^{-4}}$ *
Polynomial	18	$1.5 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$ *	$\mathbf{8 \cdot 10^{-5}}$ *
	54	$3 \cdot 10^{-3}$	$9 \cdot 10^{-4}$ *	$\mathbf{8 \cdot 10^{-5}}$ *
	96	†	$8 \cdot 10^{-4}$ *	$\mathbf{1 \cdot 10^{-4}}$ *

requires a number of points per cell, which we fix to 5000. In the setup of `i-flow`, we use $2\lceil \log_2 D \rceil$ number of coupling layers with the masking discussed in section 3.1, and the coupling transform C taken to be a Piecewise Rational Quadratic spline (appendix A.3). The neural network in each CL is taken to be a DNN of 5 layers with 32 nodes in each of the first four layers. The number of nodes in the last layer depends on the coupling transform C and the dimensionality of the integrand. For the case of Piecewise Rational Quadratic splines, the number of nodes is given by $d \cdot (3n_{\text{bins}} + 1)$, where d is the number of dimensions to be transformed. We further set the number of bins (n_{bins}) in each output dimension to 16. The learning rate was set to $1 \cdot 10^{-3}$ in all cases. We use the exponential divergence, see equation (B19), as loss function.

To compare the integrators, we set a relative uncertainty on the integral estimate as target. We then optimize the algorithms until the standard deviation of the inverse-variance weighted combination of the estimates of each optimization iteration (epoch) reaches this target. The inverse-variance weighted combination is defined as:

$$\mu = \frac{\sum_i \mu_i/\sigma_i^2}{\sum_i 1/\sigma_i^2}, \quad \sigma^2 = \frac{1}{\sum_i 1/\sigma_i^2}, \tag{24}$$

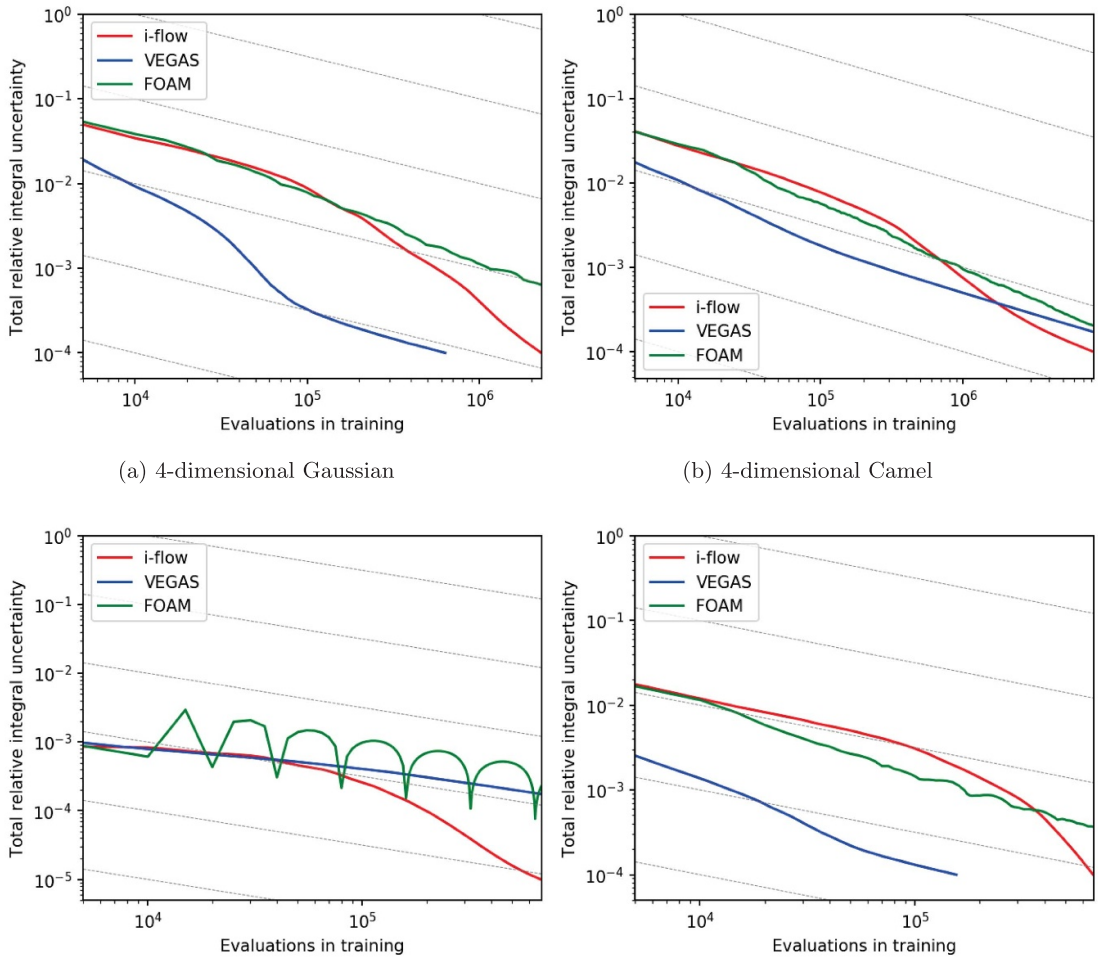
where $\mu_i(\sigma_i)$ is the mean (standard deviation) of the ith epoch and $\mu(\sigma)$ is the combination. The relative uncertainty is defined as the uncertainty of the given estimate, normalized to the true value of the integral [4]. Given this setup, there are three metrics that we use to compare the integrators: 1) the number of function calls needed to reach the target uncertainty; 2) how close the estimated integral value is to the true value; 3) the uncertainty of the estimates in the last iterations. Each of those highlights a different aspect of the integrator and we detail them below. The results are shown in tables 2–4. We chose a relative target uncertainty of 10^{-4} for the non-polynomial test functions and 10^{-5} for the polynomials. For Gaussian and Camel functions, we use $\alpha = 0.2$. In addition, we set a cut-off at $5 \cdot 10^7$ function calls.

Number of function calls. This number shows how often the integrand was evaluated by the algorithm until the target uncertainty was reached. Having fewer function calls is especially important when the function is numerically expensive to evaluate and the computational overhead of the integration algorithm becomes subleading. The results are shown in table 2. We highlight the entry with the fewest calls in boldface. In addition, we mark entries in which the final integral estimate differs by more than 5 standard deviations from the true result with a † and entries in which too much memory was required by a *.

Integral estimate and uncertainty. This obviously shows how well the integrator estimated the value of the integral. We show our results in table 3 and compare them to the true, known results. We highlight in boldface the entry with the smallest relative deviation ('pull'), defined as

$$\frac{I_{\text{code}} - I_{\text{true}}}{\sqrt{(\Delta I_{\text{code}}^2 + \Delta I_{\text{true}}^2)}}. \tag{25}$$

[4] Note that `Foam` directly gives the uncertainty including all sampled points up to the given iteration and no combination is needed.



(a) 4-dimensional Gaussian (b) 4-dimensional Camel

(c) 54-dimensional Polynomial (d) Scalar top loop

Figure 3. Accumulated (total) relative uncertainty of the integral, defined as the inverse-variance weighted combination of the uncertainties (normalized to the integral value) per iteration, as a function of number of points used in training for four different test functions. The dashed lines indicate a $1/\sqrt{N}$ scaling.

Here, I_{code} is the result from `VEGAS`, `Foam`, or `i-flow`, I_{true} is the true value of the integral, and the ΔI terms signify the uncertainty in the integral. Note, that ΔI_{true} is only non-zero for the case of the entangle circles for which it is $5 \cdot 10^{-9}$. Cases in which the cut-off for function calls was reached (see table 2) are marked with a †, cases that ran into memory problems are marked with a *.

Relative uncertainty on the integral estimate in the last iterations. The uncertainty on the integral estimate after adaptation is a measure for how well the algorithm adapted to the integrand. Once the algorithm is fully adapted, the uncertainty of a single integral estimate will be constant and the combination of all iterations will follow the $1/\sqrt{N}$ scaling law for MC estimates based on N points. A better adapted algorithm introduces a smaller coefficient for that scaling and therefore require fewer function calls to reach a smaller uncertainty. We show our results in table 4. Cases in which `VEGAS` failed to converge to the right integral value are marked with a †, a * shows entries that still showed a downward trend at the end of the optimization, indicating that the algorithm was still adapting to the integrand. We highlight the integrator with the smallest uncertainty in boldface.

For the Gaussians, `VEGAS` always has the fewest calls. This is expected, since the integrand factorizes. However, the number grows rapidly for increasing integrand dimension, whereas the number for `i-flow` grows slower. `Foam` is not able to reach the target uncertainty for $D = 8, 16$ before the cut-off of $5 \cdot 10^7$ function calls. `i-flow` has adapted best to all Gaussians of $D > 2$, as can be seen in table 4. This means that if a sufficiently small target uncertainty is required, `i-flow` would potentially need fewer function calls to reach it. The fact that the optimization of `i-flow` was not complete when the target uncertainty was reached can also be seen in figure 3(a), where the accumulated uncertainty of `i-flow` (red line) was falling quicker than $1/\sqrt{N}$ (dashed gray lines). In almost all of the cases, the integral estimate of `i-flow` was closest to the true integral value.

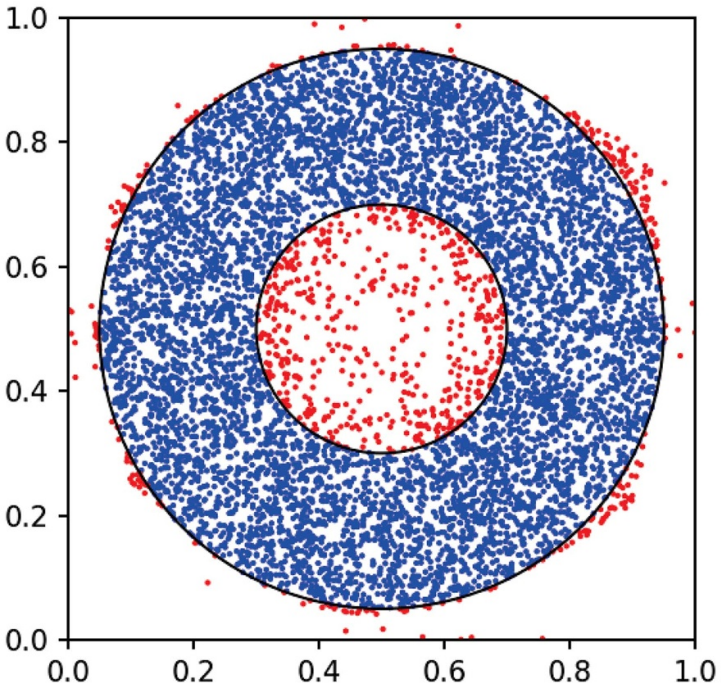


Figure 4. A set of 7500 points sampled after training `i-flow` with 5 M points on the Ring function. 6720 are inside (blue), 780 outside (red).

For the Camel functions, `VEGAS` only has the fewest calls for $D = 2$, in higher dimensions `i-flow` needs fewer calls. Note that in 16 dimensions, `VEGAS` completely misses one of the two peaks, yielding an estimate that is off by a factor of two. Since in this case the integrand is like a Gaussian, `VEGAS` converges quicker than the other algorithms. The integral estimate of `i-flow` seems also off,[5] but this is due to the fact that it needs roughly 200 epochs to 'see' the structure of the integrand and all of those iterations contribute to the final number in table 2. Again, `Foam` needs too many points for $D > 4$ to reach the target uncertainty, the integral estimates of `i-flow` are closest to the true value, and `i-flow` has adapted best to the integrand. The latter can be seen by the small relative uncertainties in table 4 and the scaling in the 4 dimensional case shown in figure 3(b).

We discuss the entangled circles and the annulus after the polynomials. For the scalar top loop, `VEGAS` needs the fewest function calls, but all 3 integrators estimate the true value within one standard deviation. It is, however, interesting to see that both `VEGAS` and `Foam` seem to be fully adapted, whereas the uncertainties of the estimates of `i-flow` were improving much faster than the $1/\sqrt{N}$ expectation, see figure 3(d).

The polynomials show the strength of `i-flow`. It has no problems adapting to the high-dimensional integrand, as can be seen in table 4. Therefore, `i-flow` needs comparatively few function calls to reach the target uncertainty. Since the polynomial does not factorize, `VEGAS` does not adapt well, or in the case of $D = 96$ not at all. The difference between the adaptation of the algorithms is also visible in figure 3(c). There, however, we see an interesting pattern in the accumulated uncertainty of `Foam` that we want to comment on. First, since `Foam` estimates the integral and uncertainty of a given iteration always on the points of all previous iterations, the uncertainty can grow for a growing number of points if the central value shifts. Second, due to the symmetry of the polynomial integrand, we see a periodic pattern that we can understand as follows. We start with an uncertainty based on the first 5000 points. Adding more points at this initial stage lets the algorithm 'see' more structure of the integrand and the uncertainty grows. A large cell with a large spread of functional values within it is then further split consecutively into many smaller cells. That reduces the spread of functional values per cell and therefore the uncertainty of the integral estimate. Once the uncertainty drops below a certain value, `Foam` stops splitting these (smaller) cells and returns to one of the 'bigger' cells it did not split in the beginning and starts splitting it. This initially increases the uncertainty again, because the spread of functional values in the large cell is larger than it was in the smaller cells. The result is the oscillating pattern we see in figure 3(c). Note that the minima of this pattern follow the $1/\sqrt{N}$ scaling.

[5]The estimate of `i-flow` only deviates from the true value because the estimates from all iterations are combined and the first 200 epochs only "see" one of the two peaks. Combining 15 of the last epochs yields 0.86377(136), which is closer to the true value.

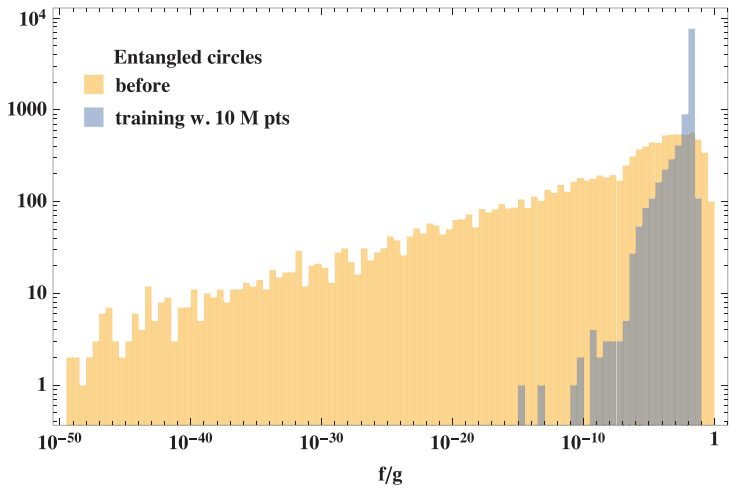


Figure 5. Weights of 10 000 points, sampled after training `i-flow` on Entangled circles (19). g is a flat distribution before training and approximately resembles the shape of f after training.

The entangled circles are best integrated by `Foam`, as it is only 2 dimensional, yet non-factorizable. `i-flow` is slightly worse, but not by much. `VEGAS`, however, does not perform well. Similar statements can be made about the annulus function with hard cuts. `VEGAS` does the worst because of the non-factorizing structure of the integrand and `Foam` does well because it is only a 2-dimensional problem. As discussed in the earlier sections, `i-flow` also allows efficient sampling once it 'learns' the integral up to small uncertainty, we therefore use these test functions to illustrate the sampling performance of `i-flow`. As an example, figure 4 shows a sample distribution after training `i-flow` with $5 \cdot 10^6$ points (1000 epochs with 5000 points per epoch) on the annulus function of equation (20). For training, we used a learning schedule with exponential decay. An initial learning rate of $2 \cdot 10^{-3}$ is halved every 250 epochs. The cut efficiency, defined as the fraction of the generated points that pass the hard cut, is 89.6%. Figure 5 shows the weights of 10 000 points sampled after training with 10 M points on the Entangled Circles of equation (19). In the ideal case of $g \to f/I$, we expect the weight distribution to approach a delta function. In figure 5, we see that the trained results are much more like a delta function than the flat prior, showing significant improvement in the ability to draw samples from this function.

It is clear from these considerations that for the low-dimensional integrals ($D \leq 4$), all three integrators achieve reasonable results. If the target uncertainty is not very small, `VEGAS` or `Foam` provide the best integrator, depending on the integrand at hand. If, however, a very small target uncertainty is needed, `i-flow` is the better option as it adapts really well to the shape of the integrand. It is only the fact that `i-flow` adapts slower than `VEGAS` that makes `i-flow` lose in the beginning, as illustrated in figure 3. For higher-dimensional integrands ($D \geq 4$) `i-flow` requires fewer function calls because it adapts better to the integrand. For example, `VEGAS` fails in the integration of 16-dimensional Camel function completely (missing one of the peaks) and `Foam` has a large uncertainty on the final result, even though it has much more function calls. `Foam` also performs poorly in the case of the Gaussian in 16 dimensions. In both of these cases, `Foam` approximately requires b^D number of cells to map out all the features of a function, where b is the average number of bins in each dimension. If b is taken to be 2, for 16 dimensions, the number of cells required is at least 2^{16}, which is far greater than our set cut-off of 10 000 cells. Therefore, when dealing with high-dimensional integrals, `Foam` is the least efficient integrator.

To quantify the computational overhead of `i-flow` in comparison to `VEGAS`, we trained both for 100 iterations with 5000 points per iteration on the polynomial function. It took `VEGAS` consistently 2 seconds for 2, 4, 8, 16, and 32 dimensions, and it took `i-flow` 14.7, 37.2, 80.1, 176.4, and 359.2 seconds, respectively, on a laptop with Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz. This increase is due to needing more Coupling Layers and therefore increasing the number of trainable parameters in higher dimensions. Working out the time for the 32 dimensions, we find that if the function evaluation takes much longer than 720 μs then the overhead starts to become unimportant. Additionally, if the difference in function evaluations to reach a target precision are taken into account, the time for function evaluation is even smaller in order for the additional overhead of `i-flow` to become insignificant.

To summarize, `i-flow` provides the best integrator for integrals in 4 or more dimensions, especially if a high precision is needed and/or the integrand is numerically expensive and slow to evaluate.

6. Conclusion and outlook

As shown in the previous section, `i-flow` tends to do better than both `VEGAS` and `Foam` for all the test cases provided. However, `i-flow` comes with a few downsides. Since `i-flow` has to learn all the correlations of the function, it takes significantly longer to achieve optimal performance compared to the other integrators. This can be seen in figure 3. This obviously translates to longer training times. Additionally, the memory footprint required for `i-flow` is much larger due to requiring storage for quicker parameter updates within the NNs. Both of these can be overcome with future improvements.

There are several directions in which we plan to improve the presented setup in the future. So far, we only used simple NN architectures in our coupling layers. Using convolutional NNs instead might improve the convergence of the normalizing flow for complicated integrands, as these networks have the ability to learn complicated shapes in images with fewer parameters than dense networks.

The setup suggested in [54] would allow the extension of `i-flow` to discrete distributions, which also has applications in HEP [52, 53]. Another way to implement this type of information is utilizing Conditional Normalizing Flows [55].

The implementation of transflow-learning, which was suggested in [56], would allow the use of a trained normalizing flow on different, but similar problems without retraining the network. Such problems arise in HEP when new-physics effects from high energy scales modify scattering properties at low energies slightly and are described in an effective field theory framework. Another application for transflow-learning would be to train one network for a given dimensionality and adapt the network for another problem with the same dimensionality.

Using techniques like gradient checkpointing [57] have the potential to reduce the memory usage substantially, therefore allowing more points to be used at each training step or larger NN architectures.

The setup presented in [58], which introduces invertible 1×1 convolutions, showed an improved performance over the vanilla implementation of the normalizing flows, which possibly also applies to our case. These 1×1 convolutions are generalizations of permutation operators acting on the inputs. Additionally, this would modify the maximum number of coupling layers required by having more expressive permutations.

Acknowledgments

We thank Joao M Goncalves Caldeira, Felix Kling, Luisa Lucie-Smith, Tilman Plehn, Holger Schulz, Nhan Tran, Paddy Fox, William Jay, David Shih, and the participants of the Aspen workshop 'The Energy Frontier Beyond the LHC Run 2' for their comments and discussions. We further thank Stefan Hóche for helpful discussions, comments, and for his `Foam` implementation [41].

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11 359 with the US Department of Energy, Office of Science, Office of High Energy Physics. This work was performed in part at Aspen Center for Physics, which is supported by National Science Foundation grant PHY-1607 611. CK acknowledges the support of the Alexander von Humboldt Foundation.

Appendix A. Coupling layer details

The implementation of the layers available in `i-flow` are detailed below. The layers are based on the work of [31, 32] and are reproduced here for the convenience of the reader.

Appendix A.1. Piecewise Linear

For the piecewise linear coupling layer [31], given K bins of width w, the probability density function (PDF) is defined as:

$$q_i(t) = \begin{cases} Q_{i1}/w & t < w \\ Q_{i2}/w & w \leq t < 2w \\ \quad\vdots & \\ Q_{iK}/w & 1 - w \leq t < 1 \end{cases} . \tag{A1}$$

The cumulative distribution function (CDF) is defined by the integral giving:

$$C(x_i^B; Q) = \alpha Q_{ib} + \sum_{k=1}^{b-1} Q_{ik}, \tag{A2}$$

where b is the bin in which x_i^B occurs ($(b-1)w \leq x_i^B < bw$), and $\alpha = \frac{x_i^B - (b-1)w}{w}$. Alternatively, we can define b as the maximal b for which $\left(C_i - \sum_{k=1}^{b-1} Q_{ik}\right) > 0$. The inverse CDF is given by:

$$x_i^B(C_i; Q) = \frac{w\left(C_i - \sum_{k=1}^{b-1} Q_{ik}\right)}{Q_{ib}} + (b-1)w. \tag{A3}$$

The Jacobian for this network is straightforward to calculate, and gives:

$$\left|\frac{\partial C}{\partial x_B}\right| = \prod_i Q_{ib}/w\,. \tag{A4}$$

The piecewise linear layers require fixed bin widths in each layer. For details on why this is required, see appendix B of [31].

Appendix A.2. Piecewise quadratic

For the piecewise quadratic coupling layer [31], given K bins with widths W_{ik}, with $K+1$ vertex heights given by V_{ik}, the PDF is defined as:

$$q_i(t) = \begin{cases} \frac{V_{i2}-V_{i1}}{W_{i1}} t + V_{i1} & t < W_{i1} \\ \frac{V_{i3}-V_{i2}}{W_{i2}} (t - W_{i1}) + V_{i2} & W_{i1} \leq t < W_{i1} + W_{i2} \\ \vdots & \\ \frac{V_{i(K+1)}-V_{iK}}{W_{iK}} \left(t - \sum_{k=1}^{K-1} W_{ik}\right) + V_{iK} & \sum_{k=1}^{K-1} W_{ik} \leq t < 1 \end{cases} \tag{A5}$$

Integrating the above equation leads to the CDF:

$$C(x_i^B; W, V) = \frac{\alpha^2}{2} (V_{ib+1} - V_{ib}) W_{ib} + V_{ib} W_{ib} \alpha + \sum_{k=1}^{b-1} \frac{V_{ik+1} + V_{ik}}{2} W_{ik}, \tag{A6}$$

where b is defined as the solution to [6] $\sum_{k=1}^{b-1} W_{ik} \leq x_i^B < \sum_{k=1}^{b} W_{ik}$, and $\alpha = \frac{x_i^B - \sum_{k=1}^{b-1} W_{ik}}{W_{ib}}$ is the relative position of x_i^B in bin b. Inverting the CDF leads to:

$$x_i^B(C_i; W, V) = W_{ib} \left(\frac{-V_{ib}}{V_{ib+1} - V_{ib}} + \sqrt{\frac{V_{ib}^2}{(V_{ib+1} - V_{ib})^2} + 2\beta} \right) + \sum_{k=1}^{b-1} W_{ik}, \tag{A7}$$

where b is defined as the solution to

$$\sum_{k=1}^{b-1} \frac{V_{ik} + V_{ik+1}}{2} W_{ik} \leq C_i < \sum_{k=1}^{b} \frac{V_{ik} + V_{ik+1}}{2} W_{ik}, \tag{A8}$$

and β is the relative position of C_i in the bin b, and is given by:

$$\beta = \frac{C_i - \sum_{k=1}^{b-1} \frac{V_{ik}+V_{ik+1}}{2} W_{ik}}{(V_{ib+1} - V_{ib}) W_{ib}}. \tag{A9}$$

Appendix A.3. Piecewise rational quadratic

For the piecewise rational quadratic coupling layer [32], given $K+1$ knot points $\left\{\left(x^{(k)}, y^{(k)}\right)\right\}_{k=0}^{K}$ that are monotonically increasing, with $(x^{(0)}, y^{(0)}) = (0,0)$ and $(x^{(K)}, y^{(K)}) = (1,1)$, and $K+1$ non-negative derivatives $\left\{d^{(k)}\right\}_{k=0}^{K}$, the CDF can be calculated using the algorithm from [59], which is roughly reproduced below.

[6] Note that this definition means $b \in [1, K]$.

First, we define the bin widths ($w^{(k)} = x^{(k+1)} - x^{(k)}$) and the slopes ($s^{(k)} = \frac{y^{(k+1)} - y^{(k)}}{w^{(k)}}$). We next obtain the fractional distance (ξ) between the two knots that the point of interest (x) lies ($\xi = \frac{x - x^{(k)}}{w^{(k)}}$, where k is the bin x lies in). The CDF is given by:

$$g(x) = \frac{\alpha(\xi)}{\beta(\xi)}, \tag{A10}$$

where the details of $\alpha(\xi)$ and $\beta(\xi)$ can be found in [59], but simplifies to:

$$g(x) = y^{(k)} + \frac{\left(y^{(k+1)} - y^{(k)}\right)\left[s^{(k)}\xi^2 + d^{(k)}\xi\left(1-\xi\right)\right]}{s^{(k)} + \left[d^{(k+1)} + d^{(k)} - 2\,s^{(k)}\right]\xi\left(1-\xi\right)}, \tag{A11}$$

which is noted to be less prone to numerical issues [59]. The inverse can be found by solving a quadratic equation [32]:

$$q(x) = \alpha(\xi) - y\beta(\xi) = ax^2 + bx + c = 0, \tag{A12}$$

where the coefficients are given in [32], solving this equation for the solution that gives a monotonically increasing x results in:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b - \sqrt{b^2 - 4ac}}, \tag{A13}$$

where the second form is numerically more precise when $4ac$ is small, and is also valid for $a = 0$ [32].

Appendix B. Loss functions

We implemented several different divergences that can be used as loss functions. They differ in $p \leftrightarrow q$ symmetry, relative weight between small and large deviations, treatment of $p = 0$ case (also in the derivative), and numerical complexity. All of them are from the class of f-divergences [37].
Pearson χ^2 divergence:

$$D_{\chi^2} = \int \frac{(p(x) - q(x))^2}{q(x)}\, dx \tag{B14}$$

Kullback-Leibler divergence:

$$D_{KL} = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \tag{B15}$$

squared Hellinger distance:

$$D_{H^2} = \int 2\left(\sqrt{p(x)} - \sqrt{q(x)}\right)^2 dx \tag{B16}$$

Jeffreys divergence:

$$D_J = \int (p(x) - q(x))(\log p(x) - \log q(x))\, dx \tag{B17}$$

Chernoff's α-divergence:

$$D_{C\alpha} = \frac{4}{1-\alpha^2}\left(1 - \int p(x)^{\frac{1-\alpha}{2}} q(x)^{\frac{1+\alpha}{2}}\, dx\right) \tag{B18}$$

exponential divergence:

$$D_e = \int p(x) \log\left(\frac{p(x)}{q(x)}\right)^2 dx \tag{B19}$$

(α, β)-product divergence:

$$D_{\alpha\beta} = \frac{2}{(1-\alpha)(1-\beta)} \int \left(1 - \left(\frac{q(x)}{p(x)}\right)^{\frac{1-\alpha}{2}}\right)\left(1 - \left(\frac{q(x)}{p(x)}\right)^{\frac{1-\beta}{2}}\right) p(x)\, dx \tag{B20}$$

Jensen-Shannon divergence:

$$D_{JS} = \frac{1}{2}\int p(x)\log\left(\frac{2p(x)}{p(x)+q(x)}\right) + q(x)\log\left(\frac{2q(x)}{p(x)+q(x)}\right) dx \tag{B21}$$

Appendix C. Sector decomposition of scalar loop integrals

Following [50], we give the integral representations of triangle and box functions in 4 dimensions using the Feynman parametrisation. To begin with, the triangle integral with external particles of energy $\sqrt{s_1}, \sqrt{s_2}, \sqrt{s_3}$ and internal propagators of masses m_1, m_2, m_3 is given by

$$\begin{aligned} I_3(s_1,s_2,s_3,m_1^2,m_2^2,m_3^2) &= \int \frac{d^4k}{i\pi^2} \frac{1}{[(k-r_1)^2-m_1^2][(k-r_2)^2-m_2^2][k^2-m_3^2]} \\ &= -\int_0^\infty d^3x\delta(1-x_{123})\frac{x_{123}^{-1}}{\mathcal{F}_{Tri}} \\ \mathcal{F}_{Tri} &= (-s_1)x_1x_3 + (-s_2)x_1x_2 + (-s_3)x_2x_3 + x_{123}(x_1m_1^2 + x_2m_2^2 + x_3m_3^2) - i\epsilon \\ x_{123} &= x_1 + x_2 + x_3 \end{aligned} \tag{C22}$$

The 3-dimensional integral is further split into 3 sectors by the decomposition:

$$1 = \Theta(x_1 > x_2, x_3) + \Theta(x_2 > x_1, x_3) + \Theta(x_3 > x_1, x_2) \tag{C23}$$

For example, when $x_3 > x_1, x_2$, after the variable transformation $t_i = x_i/x_3 (i = 1, 2)$, the integral simplifies to

$$\begin{aligned} S_{Tri}(s_1,s_2,s_3,m_1^2,m_2^2,m_3^2) &= \int_0^1 dt_1dt_2 \frac{(1+t_1+t_2)^{-1}}{\tilde{\mathcal{F}}_{Tri}(s_1,s_2,s_3,m_1^2,m_2^2,m_3^2,t_1,t_2)} \\ \tilde{\mathcal{F}}_{Tri}(s_1,s_2,s_3,m_1^2,m_2^2,m_3^2,t_1,t_2) &= (-s_1)t_1 + (-s_2)t_1t_2 + (-s_3)t_2 \\ &\quad + (1+t_1+t_2)(t_1m_1^2 + t_2m_2^2 + m_3^2) \end{aligned} \tag{C24}$$

Therefore,

$$I_3 = S_{Tri}(s_1,s_2,s_3,m_1^2,m_2^2,m_3^2) + S_{Tri}(123 \to 231) + S_{Tri}(123 \to 312) \tag{C25}$$

One can perform the same trick to treat the box integral with 4 external fields and 4 propagators. After sector decomposition, one gets

$$\begin{aligned} I_4 &= S_{Box}(s_{12},s_{23},s_1,s_2,s_3,s_4,m_1^2,m_2^2,m_3^2,m_4^2) + \textbf{permutations} \\ S_{Box}(s_{12},s_{23},s_1,s_2,s_3,s_4,m_1^2,m_2^2,m_3^2,m_4^2) &= \int_0^1 dt_1dt_2dt_3 \frac{1}{\tilde{\mathcal{F}}_{Box}^2} \\ \tilde{\mathcal{F}}_{Box} &= (-s_{12})t_2 + (-s_{23})t_1t_3 + (-s_1)t_1 + (-s_2)t_1t_2 + (-s_3)t_2t_3 \\ &\quad + (-s_4)t_3 + (1+t_1+t_2+t_3)(t_1m_1^2 + t_2m_2^2 + t_3m_3^2 + m_4^2) \end{aligned} \tag{C26}$$

ORCID iDs

Christina Gao https://orcid.org/0000-0002-8599-3966
Joshua Isaacson https://orcid.org/0000-0001-6164-1707
Claudius Krause https://orcid.org/0000-0003-0924-3036

References

[1] Hobolth A, Uyenoyama M K and Wiuf C 2008 Importance sampling for the infinite sites model *Stat. Appl. Genetics Mol. Biol.* **7** 1
[2] Mode C J *et al* 2011 *Applications of Monte Carlo Methods in Biology, Medicine and Other Fields of Science* (London: IntechOpen) C J Mode (http://doi.org/10.5772/634)
[3] Rosenbluth M N and Rosenbluth A W 1955 Monte Carlo Calculation of the Average Extension of Molecular Chains *J. Chem. Phys.* **23** 356–9
[4] MacGillivray H T and Dodd R J 1982 Monte-Carlo simulations of galaxy systems *Astrophys. Space Sci.* **86** 419–35

[5] Rogers D W O 2006 REVIEW: Fifty years of Monte Carlo simulations for medical physics *Phys. Med. Biol.* **51** R287–R301
[6] Jäckel P 2002 *Monte Carlo Methods in Finance* vol 71 (New Jersey: Wiley)
[7] Szirmay-Kalos Ló 2008 Monte Carlo Methods in Global Illumination : Photo-Realistic Rendering With Randomization (Saarbrucken, Germany: VDM Verlag)
[8] Webber B R 1986 Monte Carlo Simulation of Hard Hadronic Processes *Ann. Rev. Nucl. Part. Sci.* **36** 253–86
[9] Buckley A *et al* 2011 General-purpose event generators for LHC physics *Phys. Rept.* **504** 145–233
[10] Buckley A 2019 Computational challenges for MC event generation *19th Int. Workshop on Advanced Computing and Analysis Techniques in Physics Research: Empowering the revolution: Bringing Machine Learning to High Performance Computing (ACAT 2019) Saas-Fee, Switzerland, March 11– 15 2019* (arXiv:1908.00167)
[11] The ATLAS collaboration Computing and Software—Public Results https://twiki.cern.ch/twiki/bin/view/AtlasPublic/ComputingandSoftwarePublicResults.
[12] Frederick J E 1968 *Monte-Carlo Phase Space* (Geneva, Switzerland: CERN) p CERN-68-15
[13] Lepage G P and New A 1978 Algorithm for adaptive multidimensional integration *J. Comput. Phys.* **27** 192
[14] Lepage G P 1980 *Vegas: An Adaptive Multidimensional Integration Program* http://cds.cern.ch/record/123074
[15] Jadach S 2003 Foam: A general purpose cellular Monte Carlo event generator *Comput. Phys. Commun.* **152** 55–100
[16] Bourilkov D 2019 *Machine and Deep Learning Applications in Particle Physics* (arXiv:1912.08245)
[17] Bendavid J 2017 *Efficient Monte Carlo Integration Using Boosted Decision Trees and Generative Deep Neural Networks*(arXiv:1707.00028)
[18] Klimek M D and Perelstein M 2018 *Neural Network-Based Approach to Phase Space Integration* (arXiv:1810.11509)
[19] Otten S, Caron S, de Swart W, van Beekveld M, Hendriks L, van Leeuwen C and Podareanu D 2019 Roberto Ruiz de Austri and Rob Verheyen *Event Generation and Statistical Sampling for Physics with Deep Generative Models and a Density Information Buffer*(arXiv:1901.00875)
[20] Hashemi B, Amin N, Datta K, Olivito D and Pierini M 2019 *LHC Analysis-Specific Datasets With Generative Adversarial Networks* (arXiv:1901.05282)
[21] Sipio R Di, Giannelli M F, Haghighat S K and Palazzo S 2020 DijetGAN: A generative-adversarial network approach for the simulation of QCD dijet events at the LHC *JHEP* **08** 110 (arXiv:1903.02433)
[22] Butter A, Plehn T and Winterhalder R 2019 How to GAN LHC Events *SciPost Phys.* **7** 075
[23] Carrazza S and Dreyer Féric A 2019 *Lund jet images from generative and cycle-consistent adversarial networks* (arXiv:1909.01359)
[24] Ahdida C *et al* 2019 *Fast simulation of muons produced at the ship experiment using generative adversarial networks* (arXiv:1909.04451)
[25] Butter A, Plehn T and Winterhalder R 2019 *How to GAN event subtraction* (arXiv:1912.08824)
[26] Matchev K T and Shyamsundar P 2020 *Uncertainties Associated With GAN-Generated Datasets in High Energy Physics* (arXiv:2002.06307)
[27] Bishara F and Montull M 2019 *(Machine) Learning Amplitudes for Faster Event Generation* (arXiv:1912.11055)
[28] Dinh L, Krueger D and Bengio Y 2015 NICE: non-linear independent components estimation *3rd Int. Conf. on Learning Representations, ICLR* San Diego, CA, USA *Workshop Track Proc. 2015*
[29] Dinh L, Sohl-Dickstein J and Bengio S 2016 Density estimation using real NVP (arXiv:1912.11055)
[30] Rezende D J and Mohamed S 2015 *Variational Inference With Normalizing Flows* (arXiv:1505.05770)
[31] Müller T, Mcwilliams B, Rousselle F, Gross M and Novák J 2019 Neural importance sampling *ACM Trans. Graph.* **38** 31
[32] Durkan C, Bekasov A, Murray I and Papamakarios G 2019 Neural Spline Flows (arXiv:1906.04032)
[33] Song J, Zhao S and Ermon S 2017 A-NICE-MC: Adversarial Training for MCMC (arXiv:1706.07561)
[34] Levy D, Hoffman M D and Sohl-Dickstein J 2017 Generalizing Hamiltonian Monte Carlo with neural networks (arXiv:1711.09268)
[35] Hoffman M, Sountsov P, Dillon J V, Langmore I, Tran D and Vasudevan S 2019 NeuTra-lizing bad geometry in Hamiltonian Monte Carlo using neural transport (arXiv:1903.03704)
[36] Abadi M, *et al* 2015 TensorFlow: Large-scale machine learning on heterogeneous systems Software available from tensorflow.org (https://www.tensorflow.org/)
[37] Nielsen F and Nock R 2014 On the chi square and higher-order chi distances for approximating f-divergences *IEEE Signal Process. Lett.* **21** 10–13
[38] Takahasi H and Mori M 1974 Double exponential formulas for numerical integration *Publ. Res. Institute Math. Sci.* **9** 721–41
[39] Höche S, Prestel S and Schulz H 2019 *Simulation of Vector Boson Plus ManyJet Final States at the High Luminosity LHC* (arXiv:1905.05120)
[40] Peter Lepage G Vegas documentation release 3.4. (https://vegas.readthedocs.io/en/latest/index.html)
[41] Hoeche S Private implementation of foam integrator (https://gitlab.com/shoeche/foam)
[42] Dulat F, Mistlberger B and Pelloni A 2018 Differential Higgs production at N^3LO beyond threshold *JHEP* **01** 145
[43] Papamakarios G, Pavlakou T and Murray I 2017 Masked Autoregressive Flow for Density Estimation (arXiv:170507057)
[44] Papamakarios G, Nalisnick E, Rezende D J, Mohamed S and Lakshminarayanan B 2019 *Normalizing Flows for Probabilistic Modeling and Inference* (arXiv:1912.02762)
[45] Pinkus A 2000 Weierstrass and approximation theory *J. Approx. Theory* **107** 1–66
[46] Stone M H 1937 Applications of the theory of boolean rings to general topology *Trans. Am. Math. Soc.* **41** 375–481
[47] Stone M H 1948 The generalized Weierstrass approximation theorem *Math. Mag.* **21** 237–54
[48] Kingma D P and Adam J B 2014 *A Method for Stochastic Optimization* (arXiv:1412.6980)
[49] Wolfram Research 2019 Inc. Mathematica, Version 12.0. Champaign, IL (https://www.wolfram.com/mathematica)
[50] Binoth T, Heinrich G and Kauer N 2003 A numerical evaluation of the scalar hexagon integral in the physical region *Nucl. Phys.* **B654** 277–300
[51] Hahn T and Pérez-Victoria M 1999 Automated one-loop calculations in four and d dimensions *Comput. Phys. Commun.* **118** 153–65
[52] Gao C, Stefan Höche, Isaacson J, Krause C and Schulz H 2020 Event Generation with Normalizing Flows **101** 076002
[53] Bothmann E, Janßen T, Knobbe M, Schmale T and Schumann S 2020 Exploring phase space with Neural Importance Sampling **8** 069
[54] Tran D, Vafa K, Agrawal K K, Dinh L and Poole B 2019 Discrete Flows: Invertible Generative Models of Discrete Data (arXiv:1905.10347)
[55] Winkler C, Worrall D, Hoogeboom E, and Welling M 2019 Learning likelihoods with conditional normalizing flows (arXiv:1912.00042)

[56] Gambardella A, Baydin Alm Gş, and Torr P H S 2019 Transflow learning: Repurposing flow models without retraining (arXiv:1911.13270)
[57] Chen T, Bing X, Zhang C, and Guestrin C 2016 Training Deep Nets with Sublinear Memory Cost (arXiv:1604.06174)
[58] Kingma D P and Dhariwal P 2018 Glow: Generative Flow with Invertible 1× 1 Convolutions (arXiv:1807.03039)
[59] Gregory J A and Delbourgo R 04 1982 Piecewise rational quadratic interpolation to monotonic data *IMA J. Numer. Anal.* **2** 123–30