Gravitational wave template bank placement: A normalizing flow approach with JAX

Matriculation number ending in 5732
School of Physics and Astronomy, University of Glasgow, Glasgow, G12 8QQ, United Kingdom
(Dated: August 31, 2023)

In this report, we address the problem of optimized template bank placement for gravitational wave (GW) signal detection in the context of `Python` with `JAX` operations. We start by focusing on the GW waveforms generated by compact binaries and parameterizing the waveforms into their high-dimensional parameter spaces. By directly computing the results of Fisher information matrices (FIM) for associated GW waveforms, we establish connections between the FIM results and the Bayes factor, which approximates the standard results of a matched filter search. We then generate a target density function that can be evaluated in GW waveforms' intrinsic parameter space. The densities are consequently used to allow for normalizing flow algorithm approximations. For a parameter space with chirp mass $\mathcal{M}/M_\odot \in [1.0, 21.0]$ and symmetric mass ratio $\eta \in [0.05, 0.25]$, we found an effective FIM results generation rate of 220 iterations per second, when run on CUDA platform with 8 GB dedicated VRAM under the Windows subsystem for Linux environment with `Ubuntu`. The normalizing flow training script has yet to complete training loops to approximate the generated results at the current version of the project.

I. Introduction

Gravitational waves (GW) are the direct consequence of massive spacetime distortions propagating through the spacetime fabric at the speed of light, a prediction of Albert Einstein's theory of general relativity [1, 2]. The Laser Interferometer Gravitational-wave Observatory (LIGO) announced the discovery of the first GW detection on September 14, 2015 as GW150914 [3]. This event marks the first observation of binary black hole (BBH) mergers through GW [4]. Furthermore, LIGO and Virgo recorded the first detection of binary neutron star (BNS) merges as event GW170817 on August 17, 2017. GW170817 is subsequently observed in electromagnetic bands, marking an exciting opportunity for GW-based multi-messenger astronomy, where electromagnetic follow-up allows for simultaneous observation of the same GW event through independent measures [5, 6]. These events demonstrate that compact binary coalescence (CBC) events can be regarded as targets for GW signal detection, along with other classifications of GW signals originating from unmodeled transients, stochastic backgrounds, and periodic sources [7–9]. From the reported GW signal sources, CBC events, with combinations of black holes and neutron stars turning into BBH, BNS, and neutron star-black hole (NS-BH) mergers, are taking the majority [10–13]. Thus, this project focuses on the GW template bank placement for CBC sources.

Parameter estimation with matched filtering is one of the primary methods for GW property investigations. This effectively procedure retrieves the GW source information from the waveform itself by decoding the GW data with reasonable accuracy [14, 15]. In plain words, we fit the GW data onto some simulated and computationally inexpensive waveform templates, such that the closest match occurs at the maximum signal-to-noise ratio (SNR) [16]. This method ties deeply into Bayesian statistics, where the maximum likelihood dictates the adequacy of templates for some given GW data. Because of the unknown nature of the received GW parameter space, it is possible for a GW signal to associate with multiple templates, where the templates could also vary in the complexity of their parameter spaces. Therefore, we require a collection of templates, or a template bank, for a better match [17]. Conventional methods of matched filtering require computational costs that scale with the number of simulated templates; and by computing metric distances on each dimension, the conventional methods become computationally expensive for high dimensional parameter space analysis [18]. As a result, the placement of templates becomes crucial in the efficiency improvement of GW detection and data analysis. Therefore, we propose an alternative approach for constructing a scalable algorithm for template bank placement, where we introduce normalizing flow (NF) algorithms to approximate the geometry of the numerically calculated template densities [19]. The use of NF allows for approximation of unknown distributions by starting from a known distribution, where configurations of the known distribution are tested to minimize the Kullback–Leibler (KL) divergence through a series of differentiable and invertible operations. Essentially, NF training steps manipulate and morph a known distribution into the shape of the target distribution, where one can employ the initial, simple distribution to map out the final, complicated target distribution. [20, 21].

In this report, we first introduce the theoretical background of this project in Sec.II, with introductions to GW waveform parameters and waveform characteristics. We then move on by laying down foundational knowledge regarding detector response, Bayesian statistics, matched filtering specifics, normalising flow, and template bank placement issues. Then in Sec.III, we explain the computational implementation of the project, in its workflow and the design of data modules. Carrying on, we provide

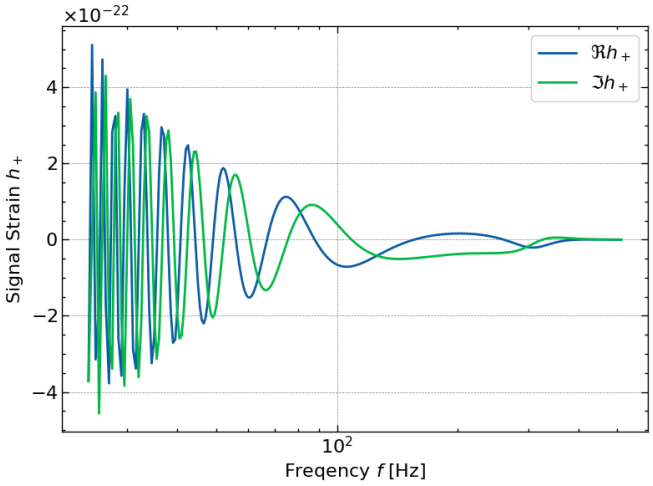

FIG. 1. Mock frequency domain GW waveform h_+ generated with `ripplegw` package, simulating `GW150914` with aligned spins. Computation requires `JAX` support. Figure shows the real and imaginary part of h_+.

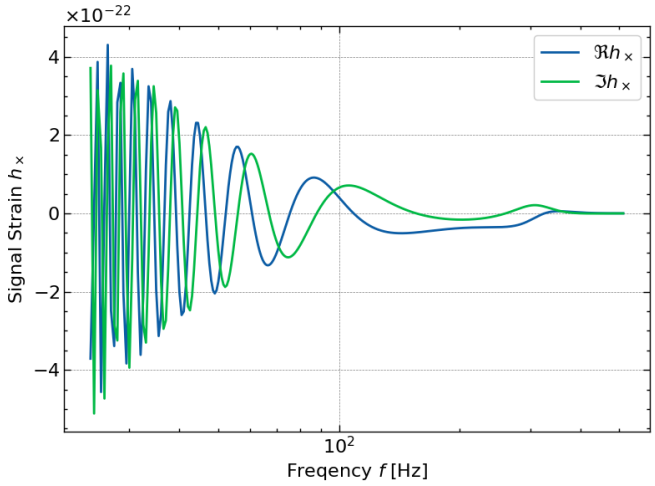

FIG. 2. Mock frequency domain GW waveform $h_\times$ generated with `ripplegw` package, simulating `GW150914` with aligned spins. Computation requires `JAX` support. Figure shows the real and imaginary part of $h_\times$.

an insight into the initial implementation of the project and the optimization techniques utilized for bringing the project to fruition. We then move on to Sec.IV, where we offer detailed explanations of the included data modules and evaluate the results generated from the project package, in its current state and potential future improvements. Finally, in Sec.V, we summarize the results of this project and provide additional comments on the work completed.

II. Theoretical background

In this section, we cover the foundational knowledge for GW waveform parameterization and subsequent calculations, the detection mechanics of GW, and a qualitative introduction to the concept of normalizing flow.

A. GW waveform and gradients

We can parameterize the GW waveforms into the corresponding intrinsic and extrinsic parameters. We demonstrate a locally generated waveform with vectorized nine-dimensional (9-D) parameter entries $\vec{\Theta}$ in Eq.(1). This technique allows for complete modelling of the GW waveform originated from a binary system, through its inspiral, merger, and ringdown (IMR) process. We see that the resulting waveform conforms with the `IMRPhenom` methods, where we achieve the waveform generation with `ripplegw` package supported by `JAX` [22, 23].

$$\vec{\Theta} = [\ \mathcal{M},\ \eta,\ s_1,\ s_2,\ d_L,\ t_c,\ \phi_c,\ \theta,\ \phi\ ]^\top \tag{1}$$

The parameters include chirp mass $\mathcal{M}$, symmetrical mass ratio η, the spins s_1, s_2 of the binary masses m_1, m_2, the distance d_L in megaparsecs, the coalescence time t_c in seconds, the phase of coalescence ϕ_c, the inclination angle θ, and the polarization angle ϕ. We therefore introduce the calculations for obtaining intrinsic parameter $\mathcal{M}$ and η in Eq.(2a) and Eq.(2b). These parameters are the direct consequences of the component masses m_1 and m_2 of CBC events. And similarly, spins s_1, s_2 are also source dependent, rendering them intrinsic as well.

$$\mathcal{M} = \frac{(m_1 m_2)^{3/5}}{(m_1 + m_2)^{1/5}} \tag{2a}$$

$$\eta = \frac{(m_1 m_2)}{(m_1 + m_2)^2} \tag{2b}$$

By focusing on these aforementioned intrinsic parameters, we generate a mock waveform and compute its gradients. The gradients are then mapped onto the frequency domain with the automatic gradient calculations performed by `JAX`. Consequently, we demonstrate the gradient calculation for some GW waveform h with $\vec{\Theta}$ in Eq.(3).

$$\tilde{h} = \frac{\partial h_{\vec{\Theta}}}{\partial \Theta}, \text{ with } \tilde{h}_i = \frac{\partial h_{\vec{\Theta}}}{\partial \Theta_i} \tag{3}$$

We can effectively generate the gradients with respect to any elements of the GW parameter. Thus, consider the polarizations of GW waveforms, where the separation of the polarizations could be utilized for computational simplifications. This means that we can mark the polarized waveforms as h_+ for plus polarization and $h_\times$ for cross-polarization. Additionally, as illustrated in Eq.(4a) and Eq.(4b), h_+ and $h_\times$ can be represented by some arbitrary amplitudes A_+ and $A_\times$, along with the time-dependent orbital phase $\Phi(t)$ [24]. The waveform and the associated gradients are shown respectively in Fig.1, Fig.2, Fig.3, and Fig.4.

$$h_+ = A_+ \cdot \cos\left(\Phi\left(t\right)\right) \tag{4a}$$

$$h_\times = A_\times \cdot \sin\left(\Phi\left(t\right)\right) \tag{4b}$$

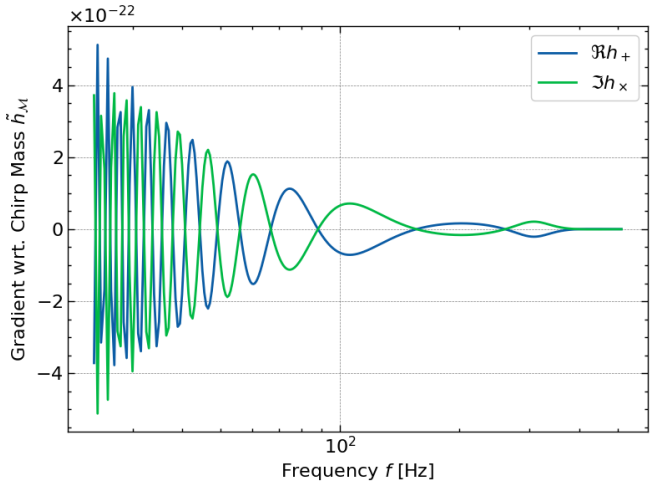

FIG. 3. Gradients of mock `GW150914` signal waveform h with respect to chirp mass $\mathcal{M}$. The gradient $\tilde{h}_{\mathcal{M}}$ is calculated from the real and imaginary parts of the GW waveforms $h_+, h_\times$ and consequently mapped onto the frequency domain. The figure indicates an aligned spin environment, where the real part of h_+ is in symmetry with the imaginary part of the $h_\times$ gradients.

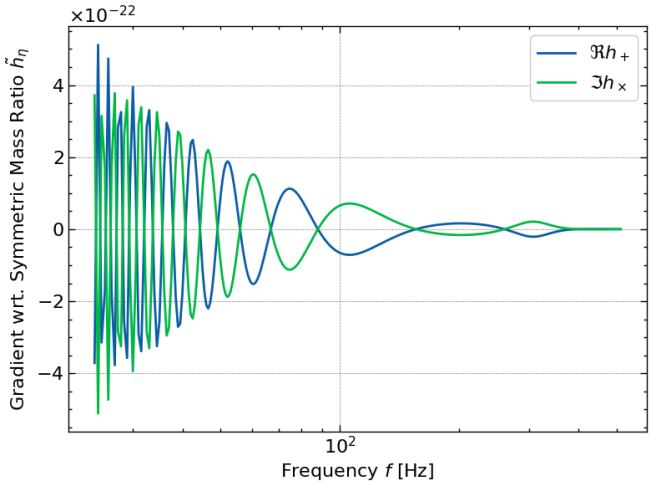

FIG. 4. Gradients of mock `GW150914` signal waveform h with respect to symmetric mass ratio η. The gradient $\tilde{h}_\eta$ is calculated from the real and imaginary parts of the GW waveforms $h_+, h_\times$ and consequently mapped onto the frequency domain. The figure indicates an aligned spin environment, where the real part of h_+ is in symmetry with the imaginary part of the $h_\times$ gradients.

That is, we observe that the polarization of GW waveforms effectively causes phase shift, where the formalism of the waveform is consistent. Therefore, we assume h_+ as the primary GW waveform formalism, as the calculations could be easily modified to suit $h_\times$ polarizations.

B. Bayesian statistics

Bayesian statistics allows for probabilistic analysis of a given model with some data. We are inferring the properties of the data by comparing it with existing theoretical models, with Bayes theorem given in Eq.(5).

$$P(\text{model}|\text{data}) = \frac{P(\text{model})P(\text{data}|\text{model})}{P(\text{data})} \tag{5}$$

Therefore, considering the probability of signal presence as S and a lack of signal, equivalent to the noise environment, as N, we could write down the likelihood ratio, or the Bayes factor, $\mathcal{B}$ for some arbitrary evidence I in Eq.(6) [25].

$$\frac{P(S|I)}{P(N|I)} = \frac{P(S)}{P(N)} \cdot \frac{p(I|S)}{p(I|N)} = \mathcal{B}(I)\frac{P(S)}{P(N)} \tag{6}$$

Moving on, the likelihood $\mathcal{L}$ of some data d matching some model h can be written as the expectation value of their inner products. We show this formalism in Eq.(7).

$$\mathcal{L}(d|h) \propto \mathbb{E}\left[-\frac{1}{2}\langle d-h|d-h\rangle\right] \tag{7}$$

That is, $\mathcal{B}$ could be rewritten as demonstrated in Eq.(8). This allows for comparisons between d and multiple h, where a better model match for the data would yield a higher $\mathcal{B}$. Consequently, we generalize d and h for the current project, where d denotes the GW waveform and h represents the waveform templates. Notice the formalism of $\langle h|h\rangle$, this normalization of waveform template gives insight on the template density, as explained in the following sections.

$$\begin{aligned}\mathcal{B} = \frac{\mathcal{L}(d|h)}{\mathcal{L}(d|0)} &= \frac{\mathbb{E}\left[-\frac{1}{2}\langle d-h|d-h\rangle\right]}{\mathbb{E}\left[-\frac{1}{2}\langle d|d\rangle\right]} \\ &= \mathbb{E}\left[\langle d|h\rangle - \frac{1}{2}\langle h|h\rangle\right]\end{aligned} \tag{8}$$

C. Waveform normalization and template density

In Eq.(9), we clarify the calculation of the noise-weighted inner products, with $\tilde{d}$, $\tilde{h}$, and the power spectral density (PSD) $S(f)$, where df and δf denote the frequency sampling interval [25, 26].

$$\begin{aligned}\langle d|h\rangle &= 4\,\mathrm{Re}\left\{\int_{f_{\min}}^{f_{\max}} \frac{\tilde{d}^*\tilde{h}}{S(f)}df\right\} \\ &= 4\delta f\,\mathrm{Re}\left\{\sum_i \frac{\tilde{d}_i^*\tilde{h}_i}{S(f)_i}\right\}\end{aligned} \tag{9}$$

The occurrence of $S(f)$ calls for the problem of noise in the received signal. Shown in Fig.5, we notice that $S(f)$ is effectively the detector response curve. The `bilby` package provides easy access for obtaining the PSD for specific detectors, such as the advanced LIGO (aLIGO) [27, 28]. That is, it is necessary to consider the variation of detector response curves when employing the calculations across different detectors, as the $S(f)$ is used to filter the

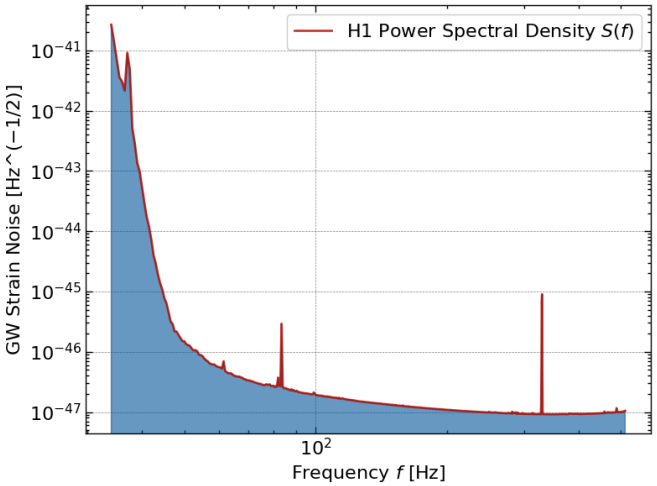

FIG. 5. Plot of $S(f)$ indexed from `bilby` package based on frequency domain of interest, base data sourced from aLIGO noise curve.

GW data and clean up the noise components. In other words, this employment of $S(f)$ is detector-dependent. As seen in Eq.(10), $S(f)$ yielded from the employment of fast Fourier transform (FFT) of the autocorrelation function C. This process is to convert the GW signal into the frequency domain, within which the waveform characteristics of a certain frequency can be examined. Mathematically, FFT allows for efficient computation, as the operations are effectively reduced from the order of $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$.

$$S_N[f] = \sum_{t=0}^{N-1} C_N[t] \cdot e^{-i2\pi f t/N} \tag{10}$$

The associated parameters f, t represent the frequency or time domain of the transformation, with N indicating the number of signal entries.

Now, we take notice that the waveform template h should follow the normalization convention addressed in Eq.(11), with $\mathcal{A}$ denoted as the normalization factor.

$$h = \mathcal{A}\hat{h}, \text{ with } \left\langle \hat{h} \middle| \hat{h} \right\rangle = 1 \tag{11}$$

Consequently, we could obtain $\mathcal{A}$ by employing Eq.(9) on the templates themselves. We demonstrate this calculation in Eq.(12).

$$\langle h|h\rangle = \left\langle \mathcal{A}\hat{h} \middle| \mathcal{A}\hat{h} \right\rangle = \mathcal{A}^2 \left\langle \hat{h} \middle| \hat{h} \right\rangle = \mathcal{A}^2 \tag{12}$$

Thus, we obtain the normalized template as seen in Eq.(13).

$$\hat{h} = \frac{h}{\mathcal{A}} = \frac{h}{\sqrt{\langle h|h\rangle}} \tag{13}$$

That is, consider the notation laid out in Eq.(9), for some data d and template h, we can rewrite the formalism with a variating normalization factor, following the matched filter notations. Shown in Eq.(14) the factor at maximum likelihood is labelled as $\mathcal{A}_{\text{ML}}$ [25].

$$\rho = \frac{\langle d|h\rangle}{\mathcal{A}} = \frac{\langle d|h\rangle}{\sqrt{\langle h|h\rangle}} = \left\langle d \middle| \hat{h} \right\rangle \tag{14}$$

This is because the maximum likelihood is manifested at $\partial_A \mathcal{L}$, as seen in Eq.(15).

$$\mathcal{A}_{\text{ML}} = \frac{\langle h|d\rangle}{\sqrt{\langle h|h\rangle}} = \left\langle \hat{h} \middle| d \right\rangle \tag{15}$$

From the normalized templates, we are effectively looking for the deviation from maximum likelihood on the parameter subspace of interest, which is a representation of a subspace metric $\tilde{g}_{ij}$ with the notations in Eq.(16) [25].

$$\tilde{g}_{ij} = \left\langle \frac{\partial \hat{h}}{\partial \Theta_i} \middle| \frac{\partial \hat{h}}{\partial \Theta_j} \right\rangle = \left\langle \partial_i \hat{h} \middle| \partial_j \hat{h} \right\rangle \tag{16}$$

This metric is the Fisher information matrix (FIM) $\Gamma_{(i,j)}$ formalism equivalent, as shown in Eq.(17) [15, 26].

$$\begin{aligned} \Gamma_{(i,j)} &= \mathbb{E}\left[\left\langle \tilde{h}_i \middle| \tilde{h}_j \right\rangle \right] \\ &= \mathbb{E}\left[4\,\mathrm{Re}\left\{ \int_{f_{\min}}^{f_{\max}} \frac{\tilde{h}_i^* \tilde{h}_j}{S(f)} df \right\} \right] \\ &= \mathbb{E}\left[4\,\mathrm{Re}\left\{ \delta f \cdot \sum \left[\frac{\tilde{h}_i^* \tilde{h}_j}{S(f)} \right] \right\} \right] \end{aligned} \tag{17}$$

Now, by correctly projecting the $\tilde{g}_{ij}$ results onto $\mathcal{M}$ and η space using Eq.(18), we obtain γ_{pq} as the projection of $\tilde{g}_{ij}$ onto ϕ_c, with $h_0, h_{\pi/2}$ being the phase-different waveforms [25].

$$\begin{aligned} \gamma_{pq} &= \tilde{g}_{pq} - \frac{\tilde{g}_{\phi_c} \tilde{g}_{q\phi_c}}{\tilde{g}_{\phi_c \phi_c}} \\ &= \left[\left\langle \partial_p \hat{h}_0 \middle| \partial_q \hat{h}_0 \right\rangle - \left\langle \partial_p \hat{h}_0 \middle| \hat{h}_{\pi/2} \right\rangle \left\langle \partial_q \hat{h}_0 \middle| \hat{h}_{\pi/2} \right\rangle \right] \end{aligned} \tag{18}$$

Going back to the time domain by projecting $\tilde{g}_{ij}$ onto t_c, we obtain the template bank with Eq.(19) [25].

$$g_{kl} = \gamma_{kl} - \frac{\gamma_{t_c k} \gamma_{t_c l}}{\gamma_{t_c t_c}} \tag{19}$$

This indicates that we have equivalently calculated results of Γ for waveform templates existing in $\mathcal{M}, \eta$ space. Thus, as seen in Eq.(20), a test statistics $\mathcal{N}$ could be seen as the template density, equivalent to the square root of the determinant of the metric, where a natural logarithm can be taken to further format the results [15, 25].

$$\mathcal{N} \propto \log\left(\sqrt{\det g} \right) \tag{20}$$

We therefore employ this test statistics as an equivalence of the template density.

D. NF formalism

The implementation of NF transforms some simple distributions, such as a uniform distribution, through a sequence of invertible and differentiable operations to approximate the results of some more complex distributions [21, 29]. The NF training process shall first generate some samples with the initial parameter of a known distribution. Subsequently, the transformations are implemented to create the best fit of a known distribution to the target distribution, where the losses are calculated as a result. From this point, the NF model shall go backwards and evaluate the gradient of the loss with respect to the proposed parameters. This step allows for the flow to locate the more influential parameters in the result of losses. Consequently, the gradients guide the update of new parameters, where a new training loop is initiated until the losses are minimized. At the end of training, one could utilize the trained parameters to approximate a target distribution from some known distributions.

In the scope of this study, we aim to incorporate NF to approximate the template bank density for 2-D parameter space of $\mathcal{M}, \eta$; and use the approximated density distributions to map out the GW template bank density. We can view the NF training process with the help of Eq.(21), where $p_z(x)$ denotes some known simple distribution and $f(x)$ denotes the sequence of transformation functions.

$$p_x\left(x\right) = p_z\left(x\right) \cdot f\left(x\right) \cdot \left|\frac{\partial f\left(x\right)}{\partial x}\right| \tag{21}$$

Thus, NF continuously updates on the conditions of f, such that the KL divergence is minimized. The KL divergence D_{KL} between a target distribution $q(x)$ and its approximation $p(x)$ with respect to some base measure $\nu(dx)$ is shown in Eq.(22).

$$D_{\mathrm{KL}}(p||q) = \int_{\chi} \left[\log \frac{p(x)}{q(x)}\right] p(x)\nu(dx) \tag{22}$$

That is, incorporating a more generic representation of GW density distribution $p(\vec{\Theta})$ and a flow sample number of N, we could rewrite Eq.(22) to obtain Eq.(23). This forward KL divergence formalism indicates that we could take the expectation value of the difference between two probabilities, which serves as a test statistic of the goodness of fit for a given model to the given data [29].

$$\begin{aligned} D_{\mathrm{KL}}(p||q) &= \int_{\chi} \left[\log \frac{p(\vec{\Theta})}{q(\vec{\Theta})}\right] p(\vec{\Theta}) d\vec{\Theta} \\ &= \mathbb{E}\left[\log p(\vec{\Theta}) - \log q(\vec{\Theta})\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\left[\log p_i(\vec{\Theta}) - \log q_i(\vec{\Theta})\right] \end{aligned} \tag{23}$$

And consequently, we present the reverse KL formalism in Eq.(24). The difference in the formalism represents the target of measure: 1) forward KL gives the information loss for using a simple distribution to approximate the complex target distribution; 2) reverse KL offers the information loss for using the target distribution to evaluate the simple distribution. In other words, KL formalism measures how exact one distribution is to the other [19].

$$D_{\mathrm{KL}}(q||p) = \frac{1}{N}\sum_{i=1}^{N}\left[\log q_i(\vec{\Theta}) - \log p_i(\vec{\Theta})\right] \tag{24}$$

As a result, the NF after training would yield a representation of the original template density distribution. We see in Eq.(25), $q(\vec{\Theta})$ can be substituted by transformed $p(\vec{\Theta})$ with some collections of parameters x.

$$q(\vec{\Theta}) \approx f(p(\vec{\Theta}), x) \tag{25}$$

Therefore, the resulting template bank densities $q(\vec{\Theta})$ could be mapped from $p(\vec{\Theta})$ with relative ease. To test for a working NF script, we now introduce the Bivirate von Mises (BVM) distribution following the formalism in Eq.(26), where k_1, k_2, k_3, μ, ν are arbitrary constants.

$$\begin{aligned} f(\phi, \psi) = {} & k_1 \cdot \cos\left(\phi - \mu\right) \\ & + k_2 \cdot \cos\left(\psi - \nu\right) \\ & - k_3 \cdot \cos\left(\phi - \mu - \psi + \nu\right) \end{aligned} \tag{26}$$

The BVM distribution serves as a functional and relatively simple high-dimensional distribution seen in 2-D parameter space. In the context of this study, we want to use a simpler distribution for conducting tests on the NF script. That is, a simpler distribution involves lighter computational costs and would in theory allow for the implementation of a more complicated, two-parameter dependent distribution such as the template bank densities.

III. Implementation

In this section, we illustrate the implementation of the software project, where the waveform, gradients, FIM, and test statistics are generated. We also provide a detailed rundown of the optimization process used to improve computational performance from the initial implementation. Afterwards, we move on to the implementation of NF for target GW template bank density approximation.

A. Waveform handling, gradients, and FIM

The waveforms are generated with the parameters specified in Eq.(1). However, the `ripplegw` package first utilizes the parameter entry of m_1, m_2. This formalism causes a back-and-forth copying of parameters and redundant calculation of $\mathcal{M}, \eta$. By directly importing the waveform parameters into the `ripplegw` package, we could simply draw out the range of $\mathcal{M}$ and η in interest, and directly create a repository of waveform parameter $\vec{Theta}$. These $\vec{\Theta}$ are then passed into the pipeline to

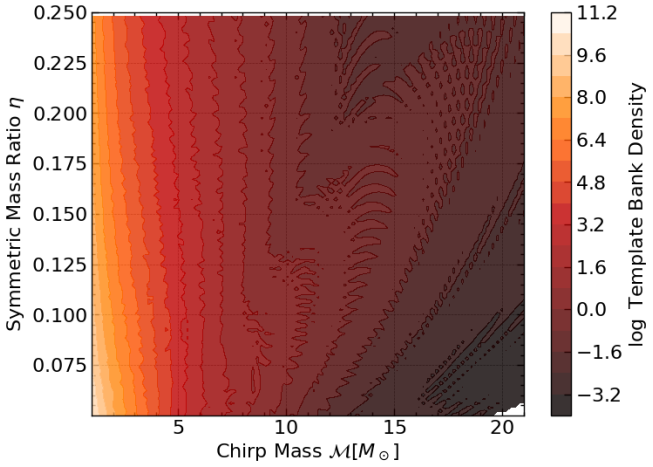

FIG. 6. Projected template density of h_+ polarization related results with log base. Plot generated on intrinsic parameter space of $\mathcal{M}, \eta$, with $\mathcal{M} \in [1.0, 21.0]$.

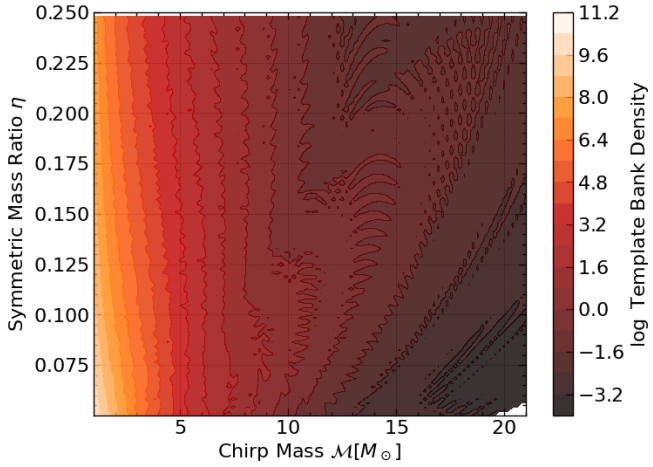

FIG. 7. Projected template density of $h_\times$ polarization related results with log base. Plot generated on intrinsic parameter space of $\mathcal{M}, \eta$, with $\mathcal{M} \in [1.0, 21.0]$.

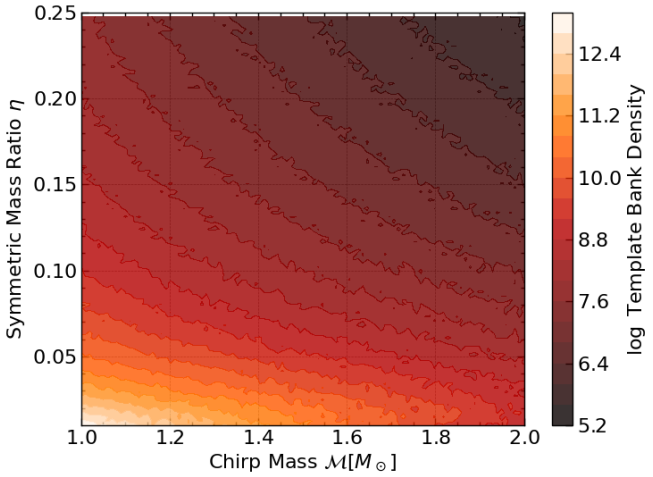

FIG. 8. Projected template density of h_+ polarization related results with log base. Plot generated on intrinsic parameter space of $\mathcal{M}, \eta$, with $\mathcal{M} \in [1.0, 2.0]$.

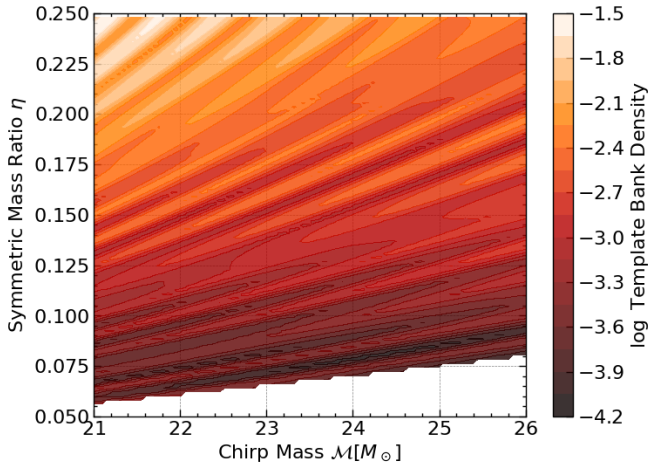

FIG. 9. Projected template density of h_+ polarization related results with log base. Plot generated on intrinsic parameter space of $\mathcal{M}, \eta$, with $\mathcal{M} \in [21.0, 26.0]$.

generate their associated waveforms, gradients, and subsequent densities with FIM.

Notice that, by gaining access to the `jax.vmap()` and `jax.grad()` function in `JAX`, we could map the waveform gradients onto the frequency domain of our interest. A sample waveform following the component mass values of GW150914 is demonstrated in Fig.1 and Fig.2, with the corresponding gradients on $\mathcal{M}$ and η in Fig.3 and Fig.4. Now, because of the parallel nature of GPU usage, `JAX` methods map the entire gradients simultaneously. This may cause a delay in the resulting output, as not all the parameters are of interest to this project.

Moving on, the Γ is first computed by list comprehension methods, where we obtain the determinant of the matrix one at a time. This implementation served as a temporary placeholder to test for the validity of the algorithm and was quickly replaced by vectorized calculations. The vectorization is achieved by establishing a meshgrid of $\mathcal{M}$ and η entries with `jax.numpy` aliased as `jnp`, which is later mapped onto the frequency domain by employing `jax.vmap()` method. That is, we simply get all the Γ results in one run and reshape the densities into our desired shape. The second implementation is not without problems, as the size of the grid of the parameter entries becomes significant, the memory required to complete the calculations becomes taxing. Therefore, a third iteration is introduced to utilize the compilation optimization of `jax.jit()` as well as packaged Γ projection with list comprehension. Regardless of the approach, seen in Fig.5, $S(f)$ remains constant throughout the implementation, as the targeted frequency range was not altered.

With the corrected projection of Γ onto ϕ_c, t_c, we achieved the template density as seen in Fig.6 and Fig.7.

The results are consistent with the targeted template densities on m_1, m_2 parameter subspace that we aim to recreate, with higher template bank densities seen in lower $\mathcal{M}$ and lower η [25].

B. Compilation optimization and data structures

The `JAX` package is a form of accelerated linear algebra (XLA) that allows for accelerated and parallel computation of data arrays [23]. There are three major notes to take for the usage of `JAX`: (1) the ability to use compilation cache for handling array inputs of the same shape; (2) the ability to automatically calculate the gradients of some function and parallelly map the results onto some other array; (3) the placement of `@jax.jit()` decorator for efficient usage of overhead compilation, where the `@jax.jit()` decorator should be placed at the outermost layer of function calls. The use of `JAX` is especially useful for the case of this project, as the parameters passed into the calculation pipelines could be adjusted to follow the same shape for improved efficiency. In essence, we seek to generate a collection of waveform parameter $\vec{\Theta}$ and obtain the corresponding FIM results.

The use of matrices in the program makes it so that the computations are performed simultaneously. This massive parallelism calls for significant memory allocations. One approach of circumventing the out-of-memory error is to batch the calculations, such that only a part of the data set is processed at a time. During the generation of $\vec{\Theta}$, we chose the number of parameter entries for $\mathcal{M}, \eta$ at n entries each, giving both arrays a shape of `(`n`, )`. The resulting meshgrid size can therefore be presented as `(`n^2`, 2)` for a passed-in 2-D parameter of $\mathcal{M}, \eta$. That is, when expanding to higher dimension parameter entries, we yield a shape of `(`n^2`, num_param)` for `num_param` indicating the number of active parameters used to substitute out the base parameter vector $\vec{\Theta}$. In the context of this study, because of the given complexity of the density calculations, we split the $\vec{\Theta}$ array apart and obtained the results in sequence. After which, the results are concatenated to produce the final density array. With the introduction of a compilation cache using `jax.jit()`, the batched calculation time remains similar to that of a memory-intensive process. Recall that `jax.jit()` requires array entries to be of the same shape. We therefore could manually implement the first compilation by passing in some waveform parameters to allow for initial Γ calculation. Additionally, a persistent cache could be locally established to allow `JAX` to skip compilation for specific shape entries when being executed in the same instance, given that the shape has already been compiled locally.

Aside from batching, we also noticed that the target parameter array $\vec{\Theta}$ contains multiple entries that are not of the current scope of interest. We can therefore use some base parameter $\vec{\Theta}$ to fill in the blanks at irrelevant indices. For instance, since we are interested in the template bank density correlation with $\mathcal{M}$ and η, we could establish a fixed array entry for $\vec{\Theta}$ at python array index `[2:]`, where we simply insert the values of $\mathcal{M}$ and η into the first two indices and construct a $\vec{\Theta}$ for waveform generation process. This simplification is further manifested in the gradient calculations part, where the unnecessary mappings of gradients from entries such as d_L and s_1, s_2 are skipped.

C. Flow incorporation

The NF script is established by first creating classes for target densities. This step allows for verification that the flow is in a working state while gating the complex calculations before the final implementation is needed. We start with testing a simple BVM distribution, which offers foundations for testing out target distributions based on 2-D parameter entries. Within the target distribution class, we create functions that calculate the target distribution as well as the natural logarithm-based distribution values. We then use random number generators to create initial parameter entries to be passed into the known distribution. This is effectively the sampling process. By calculating the KL divergence at the specific past in arrays, NF updates the parameter guesses for the future iteration. This is equivalent to a morphological change in the known distribution, causing it to become one step closer to the shape of the target distribution. In essence, to allow for NF to learn the shape of the target distribution, we rely on the implementation of `haiku` with `flax`, `distrax`, `optax` based on the `JAX` ecosystem [30–32].

IV. Data modules and analysis

We now give a detailed rundown of the specific data modules created, their usage and purpose, as well as how the modules interact with each other. We began with establishing a working directory, where we created a subdirectory to manage the data modules of this project. An `__init__.py` file is present to package the utilities of the data folder together. We then present an analysis of the NF training, where the initial testing of a simple distribution provides functional proof of the script.

Additionally, a persistent cache folder for `JAX` is located within the working directory. Notice that most, if not all, aspects of this project are written with functional programming and modularity in mind. Detailed comments and docstrings are provided to support future development, along with a `README.md` file to offer additional instructions for executing the program. We also included a working licensing file `LICENSE` for future use.

A. Module file: `gw_cfg.py`

To begin with, we created a configuration file named `gw_cfg.py`. This configuration file manages the usage of parameters that are of interest to the investigation. The configuration file is responsible for holding the constants, such as the base $\vec{\Theta}$ array at $\vec{\Theta} = [\,\mathcal{M},\ \eta,\ 0.0,\ 0.0,\ 40.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0\,]^\top$, as well as generating the input parameter arrays. Since this is the sole configuration file of the project, we took the liberty to further simplify the external function calls by directly computing the global variables and have them imported

for other modules to reference.

B. Module file: `gw_rpl.py`

In this file, we host a collection of functions that generate the mock GW waveform according to the waveform parameter $\vec{\Theta}$ with `ripplegw`'s `IMRPhenom` method. These waveforms are then normalized with a normalization function following Eq.(13). In the end, another function takes the gradients of the waveform and maps them onto the frequency domain with `jax.vmap()`.

Additionally, these functions are reliant on the one-sided noise-weighted inner product addressed in Eq.(9), while being separated for polarized waveforms. Note that, even though the operations performed on the waveform polarization of interest, h_+, are identical for cross-polarized $h_\times$, `JAX` encourage separate, top-level handling of `@jax.jit` decorators. In practice, we could further optimize the top-level function call to allow for the incorporation of the different waveforms under a single function call.

C. Module file: `gw_fim.py`

This is the primary calculation module of the entire project, namely, the functions required to produce the template bank density are housed within. A function for assigning the elements of a basic FIM is first introduced. As seen in Eq.(9), we need to first obtain the gradients for the waveform template in interest. Afterwards, we employ a new function to get the mapped gradients for building the foundation of FIM results.

We then create two more functions, where the FIM is projected onto ϕ_c and t_c in sequence, following the notations in Eq.(18) and Eq.(19). These projection functions allow for a wrapped function call, where the projections are called to correct the foundation of the FIM result. In effect, we have now obtained the method to generate a correctly projected FIM from some basic FIM that is reliant on the initial parameter entry $\vec{\Theta}$.

In the end, we take the square root of the determinant of the projected FIM to obtain the test statistics, a value of template density equivalence, following Eq.(20). Afterwards, we invoke batching methods to compute many template density results in sequence while taking advantage of the parallel computation and shape-optimized operations from `@jax.jit` enabled function calls. With the use of batching and mapping tools from `jax`, we are able to rapidly calculate the template density for any valid $\mathcal{M}, \eta$ combination at $t_c = \phi_c = 0.0$ at an equivalence of 220 iterations per second. The results at different $\mathcal{M}$ ranges are explored and laid out in Fig.8 and Fig.9.

D. Module file: `gw_plt.py`

Carrying on from all the preceding data modules, we create a separate file to manage the plot functions. These plot functions allow for easier access to plot testing, where predefined plot styles and customization arguments are present. The functions also save the figures to their respective local directories.

To further simplify the styling component of the plots, we employ `SciencePlots`. Further improvement to the plotter module could include: 1) an automated file name assignment for each plot generation; 2) an overall optimization of script placements to minimize redundant and repetitive function calls; 3) and a global label dictionary and a keyword argument repository, for easier labelling and plot style configuration.

E. Module file: `vi_cfg.py`

This configuration file contains the constants used to define the NF parameters such as a pseudo-random number generator key, the flow training epochs, flow samples, and learning rate. This file also helps to set up specific target distributions by calling the said distribution with configured class arguments. Similar to the configuration file employed for GW template bank density generation, this configuration file acts as a repository that passes on the argument to other modules in the directory.

F. Module file: `vi_cls.py`

Now, we host the target distribution classes within this module. Since classes are independent of each other, we could theoretically incorporate a uniformly formatted class structure to allow for simple switches between target distributions. In the scope of this project, we included BVM distribution along with the template bank density classes, with `log_prob()` and `prob()` methods to obtain: 1) the natural logarithm-based probability; 2) and the probability itself through an exponential calculation of the logarithmic probability.

G. Module file: `vi_dat.py`

This is the primary script for NF calculations, where a `Python` construct can be included to allow for direct execution of the NF training script. The script also includes the functions used for sampling the randomly generated parameter entries, along with the functions to update, morph, and train the NF. A conditioner function is created, where a neural network would work to parameterize the spline. Following on, an NF model is introduced with a nested bijector function that works for rational quadratic spline. Within the NF model, an alternating binary mask is created to fit to the input shape, after which, the NF is inverted.

These operations allow for forward and backward transformation of the sample parameter entry to the output of the NF while retaining the ability to trace back to the original sample parameter entries from its output. Continuing on, we now create some functions to allow for: 1) the NF to sample and compute the natural logarithm-based probabilities; 2) the NF to calculate the probability density from sampled entries; 3) the loss function to obtain the KL divergence through Eq.(22); 4) and the update function to employ stochastic gradient descent, where the sampled parameters are updated by an optimizer.

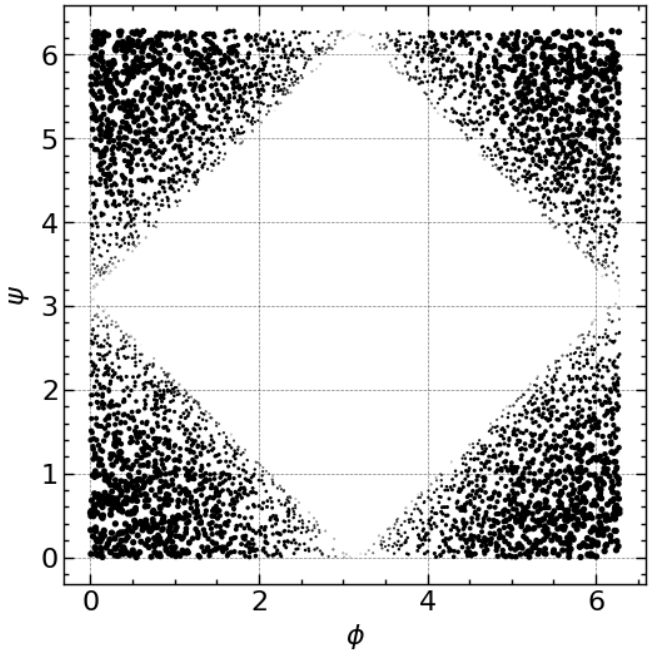

FIG. 10. BVM distribution as calculated with random number entries on ϕ, ψ parameter space where $\phi \in [0, 2\pi]$ and $\psi \in [0, 2\pi]$. Marker size is directly proportional to the distribution value.

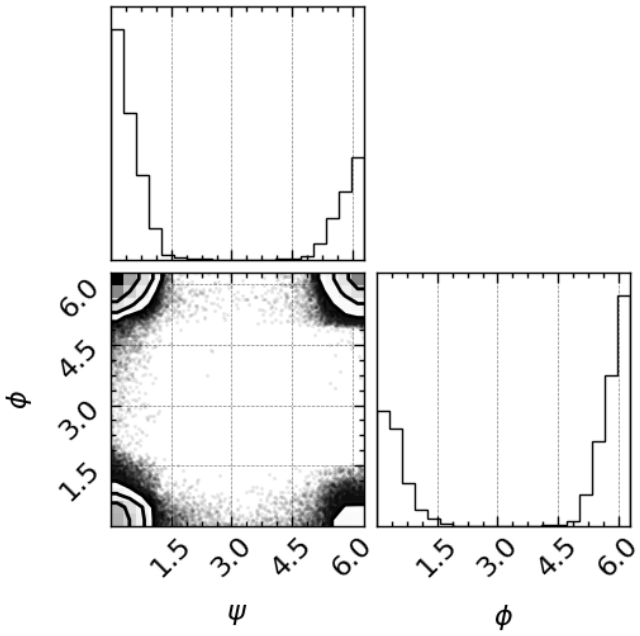

FIG. 11. BVM distribution as approximated with NF model on ϕ, ψ parameter space where $\phi \in [0, 2\pi]$ and $\psi \in [0, 2\pi]$. NF training was performed with 1000 samples at a learning rate of 0.001 over 3000 epochs.

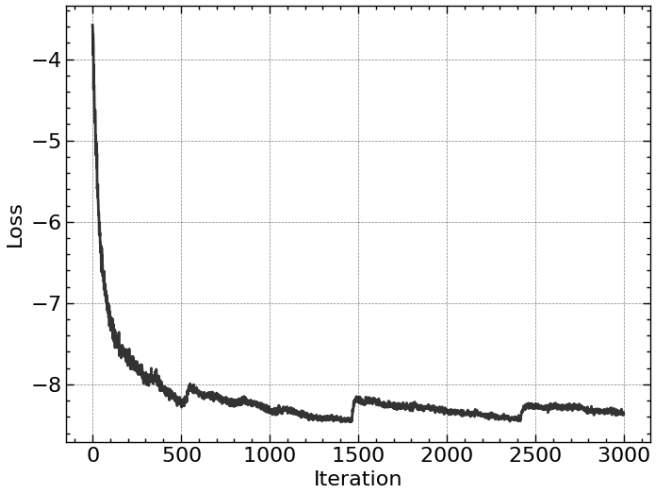

FIG. 12. The training loss across training epochs. Training performed with 1000 samples with a learning rate of 0.001. The loss descends and smooths out over 3000 training epochs.

H. Module file: `vi_plt.py`

By convention, we now gather the individual plot functions to plot the NF results. We included a `.gif` file maker to observe the changes in parameter distributions throughout training epochs. The training loss and the final posterior distribution are also callable from this module.

I. NF training and results

We can now start the NF training process by directly calling the training function from a testing script, where the results would be concatenated and plotted to: 1) obtain a posterior distribution of the approximated densities against the input parameters; 2) observe the losses at Different in what way?different training epochs; 3) and an animated posterior distribution as NF is getting trained. These specifications have been laid out in the plotting module. That said, we start by passing in the global constants that assemble the target distribution. Afterwards, a pseudo-random number generator is defined along with the optimizer. We perform an initial test on the NF script with a BVM distribution, following the formalism in Eq.(26). The BVM distribution is shown in Fig.10, with plots generated by uniformly random number generators for $\psi \in [0, 2\pi]$ and $\phi \in [0, 2\pi]$. The training follows these steps: 1) generate random samples of parameter entries; 2) obtain the losses between the sampled NF model density result and the result from the target density; 3) morph and update the NF model for the next iteration. The training losses from and printed out at every 100th epoch. After 3000 epochs of training, the NF yields the loss curve and final posterior approximation on BVM, as laid out in Fig.12 and Fig.11 respectively. In all, we recognize that the NF training is working successfully with relatively simpler distributions that can be evaluated in 2-D parameter space.

When attempting to incorporate the template density into the NF script, we encountered unknown errors that

resulted in the process crashing after significant running time. Initial debugging focus on the array shape and data type. After verifying the functions are for vectorized operations, we moved on to the investigations on `@jax.jit` placements across the project file. By running the batched commands for template density calculations, we observed minimal difference in the redundant placement of `@jax.jit` decorators. However, we removed the decorators to conform to the recommended placement of `@jax.jit` as laid out in the documentation. Next, we attempted to add additional print statements to the training loop to identify the line where the training grinds to a halt. By running two trials with minimal sample numbers, thus mitigating the use of batching in the calculation, we observe no difference in the code execution. Further debugging located the error to the `update()` function, where the gradient calculation of the loss function appears to be problematic. We suspect that this has something to do with the complex nature of the template density, where `JAX` compilation may have triggered a runtime issue during the compilation due to numerical instability such as `NaN` occurrences, causing the process to be killed down the line. Given time, further investigation may resolve this issue, as the internal calculations used for generating template bank density have been evaluated to allow for efficiency loss function computation.

V. Conclusion

In this project, we have explored and developed an efficient FIM calculation `Python` program for approximating GW template bank density on the intrinsic parameter space of $\mathcal{M}, \eta$. We employed the automatic gradient calculation and the function vectorization methods of `JAX` to achieve rapid and low memory cost computations of template bank density. We used the third-party package `ripplegw` to generate waveform templates from 9-D parameters. The template density calculation program was tested and optimized to achieve high computational efficiency in memory cost and computational time.

We then explore the implementation of NF training for template bank density approximation by first examining the NF training script on the simpler BVM distribution in 2-D space. We noticed that the script used for loss yields incorrect results, as the KL divergence continuously approached a negative value as if it was converging on zero. However, it does not appear to affect the training results. Afterwards, we attempted to implement the NF script with template density. This attempt led to the discovery of problematic flow handling of the currently implemented template density calculation, where the optimizer is unable to update the parameter entries. An error in the gradient calculation of the loss function caused this issue in the parameter update process. The key takeaway for this project is that the evidence of successful 2-D approximation of BVM distribution has pointed to the potential of implementing `JAX` supported NF script for learning the shape of complex GW template bank density in its parameter space, where further investigations are needed to resolve pending issues in the current project stage.

Acknowledgments

We thank Dr. John Veitch, School of Physics & Astronomy, University of Glasgow for dedicated supervision and enlightening guidance that brought this project to life. We thank Dominika Zieba, School of Physics & Astronomy, University of Glasgow for active assistance on machine learning algorithms involving normalizing flows and its subsequent code integrations. We also thank the members of the Institute for Gravitational Research, School of Physics & Astronomy, University of Glasgow for inspiring seminars and intriguing discussions.

[1] A. Einstein, Annalen der Physik **354**, 769 (1916), URL https://doi.org/10.1002/andp.19163540702.

[2] A. Einstein, Sitzungsberichte der Königlich Preussischen Akademie der Wissenschaften pp. 688–696 (1916).

[3] B. P. Abbott, R. Abbott, R. Adhikari, P. Ajith, B. Allen, G. Allen, R. S. Amin, S. B. Anderson, W. G. Anderson, M. A. Arain, et al., Reports on Progress in Physics **72**, 076901 (2009), URL https://doi.org/10.1088/0034-4885/72/7/076901.

[4] B. P. Abbott, R. Abbott, T. D. Abbott, M. R. Abernathy, F. Acernese, K. Ackley, C. Adams, T. Adams, P. Addesso, R. X. Adhikari, et al., Physical Review Letters **116** (2016), URL https://doi.org/10.1103/physrevlett.116.061102.

[5] F. Acernese, M. Agathos, K. Agatsuma, D. Aisa, N. Allemandou, A. Allocca, J. Amarni, P. Astone, G. Balestri, G. Ballardin, et al., Classical and Quantum Gravity **32**, 024001 (2014), URL https://doi.org/10.1088/0264-9381/32/2/024001.

[6] B. P. Abbott, R. Abbott, T. D. Abbott, F. Acernese, K. Ackley, C. Adams, T. Adams, P. Addesso, R. X. Adhikari, V. B. Adya, et al., Physical Review Letters **119** (2017), URL https://doi.org/10.1103/physrevlett.119.161101.

[7] B. Abbott, R. Abbott, R. Adhikari, J. Agresti, P. Ajith, B. Allen, R. Amin, S. B. Anderson, W. G. Anderson, M. Arain, et al., Physical Review D **76** (2007), URL https://doi.org/10.1103/physrevd.76.042001.

[8] B. P. Abbott, R. Abbott, R. Adhikari, P. Ajith, B. Allen, G. Allen, R. S. Amin, S. B. Anderson, W. G. Anderson, M. A. Arain, et al., Physical Review D **79** (2009), URL https://doi.org/10.1103/physrevd.79.122001.

[9] R. Abbott, T. D. Abbott, F. Acernese, K. Ackley, C. Adams, N. Adhikari, R. X. Adhikari, V. B. Adya, C. Affeldt, D. Agarwal, et al., Astronomy & Astrophysics **659**, A84 (2022), URL https://doi.org/10.1051/0004-6361/202141452.

[10] A. H. Nitz, C. Capano, A. B. Nielsen, S. Reyes, R. White, D. A. Brown, and B. Krishnan, The Astrophysical Jour-

nal **872**, 195 (2019), URL https://doi.org/10.3847/1538-4357/ab0108.

[11] A. H. Nitz, T. Dent, G. S. Davies, S. Kumar, C. D. Capano, I. Harry, S. Mozzon, L. Nuttall, A. Lundgren, and M. Tápai, The Astrophysical Journal **891**, 123 (2020), URL https://doi.org/10.3847/1538-4357/ab733f.

[12] A. H. Nitz, C. D. Capano, S. Kumar, Y.-F. Wang, S. Kastha, M. Schäfer, R. Dhurkunde, and M. Cabero, The Astrophysical Journal **922**, 76 (2021), URL https://doi.org/10.3847/1538-4357/ac1c03.

[13] A. H. Nitz, S. Kumar, Y.-F. Wang, S. Kastha, S. Wu, M. Schäfer, R. Dhurkunde, and C. D. Capano, *4-ogc: Catalog of gravitational waves from compact-binary mergers* (2021), URL https://arxiv.org/abs/2112.06878.

[14] J. Veitch, V. Raymond, B. Farr, W. Farr, P. Graff, S. Vitale, B. Aylott, K. Blackburn, N. Christensen, M. Coughlin, et al., Physical Review D **91** (2015), URL https://doi.org/10.1103/physrevd.91.042003.

[15] C. Cutler and É. E. Flanagan, Physical Review D **49**, 2658 (1994), URL https://doi.org/10.1103/physrevd.49.2658.

[16] S. Khan, K. Chatziioannou, M. Hannam, and F. Ohme, Physical Review D **100** (2019), URL https://doi.org/10.1103/physrevd.100.024059.

[17] I. W. Harry, B. Allen, and B. S. Sathyaprakash, Physical Review D **80** (2009), URL https://doi.org/10.1103/physrevd.80.104014.

[18] B. J. Owen and B. S. Sathyaprakash, Physical Review D **60** (1999), URL https://doi.org/10.1103/physrevd.60.022002.

[19] D. J. Rezende and S. Mohamed, *Variational inference with normalizing flows* (2015), URL https://arxiv.org/abs/1505.05770.

[20] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan (2019), URL https://arxiv.org/abs/1912.02762.

[21] I. Kobyzev, S. J. Prince, and M. A. Brubaker, IEEE Transactions on Pattern Analysis and Machine Intelligence **43**, 3964 (2021), URL https://doi.org/10.1109/tpami.2020.2992934.

[22] T. Edwards, *ripple: Differentiable gravitational waveforms with JAX* (2023), URL https://github.com/tedwards2412/ripple.

[23] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, et al., *JAX: composable transformations of Python+NumPy programs* (2018), URL http://github.com/google/jax.

[24] C. V. D. Broeck, D. A. Brown, T. Cokelaer, I. Harry, G. Jones, B. S. Sathyaprakash, H. Tagoshi, and H. Takahashi, Physical Review D **80** (2009), URL https://doi.org/10.1103/physrevd.80.024009.

[25] T. Dent and J. Veitch, Physical Review D **89** (2014), URL https://doi.org/10.1103/physrevd.89.062002.

[26] D. Finstad and D. A. Brown, The Astrophysical Journal **905**, L9 (2020), URL https://doi.org/10.3847/2041-8213/abca9e.

[27] G. Ashton, M. Hübner, P. D. Lasky, C. Talbot, K. Ackley, S. Biscoveanu, Q. Chu, A. Divakarla, P. J. Easter, B. Goncharov, et al., The Astrophysical Journal Supplement Series **241**, 27 (2019), URL https://doi.org/10.3847/1538-4365/ab06fc.

[28] T. L. S. Collaboration, J. Aasi, B. P. Abbott, R. Abbott, T. Abbott, M. R. Abernathy, K. Ackley, C. Adams, T. Adams, P. Addesso, et al., Classical and Quantum Gravity **32**, 074001 (2015), URL https://doi.org/10.1088/0264-9381/32/7/074001.

[29] M. J. Wainwright and M. I. Jordan, Foundations and Trends® in Machine Learning **1**, 1 (2007), URL https://doi.org/10.1561/2200000001.

[30] T. Hennigan, T. Cai, T. Norman, L. Martens, and I. Babuschkin, *Haiku: Sonnet for JAX* (2020), URL http://github.com/deepmind/dm-haiku.

[31] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee, *Flax: A neural network library and ecosystem for JAX* (2023), URL http://github.com/google/flax.

[32] I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, et al., *The DeepMind JAX Ecosystem* (2020), URL http://github.com/deepmind.

A. Documentation: README.md

````
# gla-igr-msc-project

This is the degree project for *MSc in Astrophysics* at *University of Glasgow*.

- Initialized: May 30, 2023
- Edited: August 30, 2023

[//]: # "============================================================================="

## Purpose

Use normalizing flow for approximating gravitational wave template bank density

### Working scripts

- Template density for polarized GW waveform with respect to $\mathcal{M}$ and $\eta$
- Normalizing flow script - *under maintenance*

### Work in progress

- Normalizing flow script
- NF script packaging into vi_* files

[//]: # "============================================================================="

## Requirements

### Environment

- ```WSL: Ubuntu```
- ```Python 3.10```

### Dependecies

- ```jax```
- ```ripplegw```
- ```bilby```
- ```haiku```
- ```distrax```
- ```optax```
- ```scienceplots```

[//]: # "============================================================================="

## Theory background

### Waveform parameters

```python
# GW150914 Mock Param
m1, m2, s1, s2, dl, tc, phic, theta, phi = (
    36.0, 29.0, 0.0, 0.0, 40.0, 0.0, 0.0, 0.0, 0.0,
)
# Param for ripple waveform generation
mc, eta, s1, s2, dl, tc, phic, theta, phi = (
    28.0956, 0.2471, 0.0, 0.0, 40.0, 0.0, 0.0, 0.0, 0.0,
)
````
````

````
```

[//]: # "============================================================================="

## File structure

```bash
.
|-- LICENSE
|-- README.md
|-- data
|   |-- __init__.py
|   |-- __jaxcache__
|   |-- __pycache__
|   |-- gw_cfg.py
|   |-- gw_fim.py
|   |-- gw_plt.py
|   |-- gw_rpl.py
|   |-- vi_cfg.py
|   |-- vi_cls.py
|   |-- vi_dat.py
|   |-- vi_plt.py
|-- figures
|-- legacy
|-- main.py
|-- test.py
|-- results
```

[//]: # "============================================================================="

## Sample run

```python
"""
This is the master script for MSc project.

Created on Thu August 03 2023
"""
# Library import
# Set XLA resource allocation
import os
# Use jax and persistent cache
from jax.experimental.compilation_cache import compilation_cache as cc
# Custom packages
from src.template_flow import gw_fim, gw_plt, gw_rpl, vi_dat
from src.template_flow.gw_cfg import MCS, ETAS, PARAM_TEST, F_SIG, F_PSD
# Setup
os.environ['XLA_PYTHON_CLIENT_PREALLOCATE'] = 'false'
cc.initialize_cache("./data/__jaxcache__")

# First compilation test for sub modules
# Wavefor generation
HP = gw_rpl.waveform_plus_restricted(PARAM_TEST, F_SIG)
HC = gw_rpl.waveform_cros_restricted(PARAM_TEST, F_SIG)
# Gradient calculation
GP = gw_rpl.gradient_plus(PARAM_TEST)
GC = gw_rpl.gradient_cros(PARAM_TEST)
````

```
# FIM test statistics calculation
DETP = gw_fim.log_sqrt_det_plus(PARAM_TEST)
DETC = gw_fim.log_sqrt_det_cros(PARAM_TEST)
# First compilation - results checker
print(f"Test waveform HP.shape:{HP.shape} hc.shape:{HC.shape}")
print(f"Test gradient gp.shape:{GP.shape} gc.shape:{GC.shape}")
print(f"Test log density detp:{DETP:.4g} detc:{DETC:.4g}")

# FIM density calc params
FIM_PARAM = gw_fim.fim_param_build(MCS, ETAS)
print(f"fim_param.shape:{FIM_PARAM.shape}")

# New compilation for vectorized operaions
DENSITY_P = gw_fim.log_density_plus(FIM_PARAM).reshape([len(MCS), len(ETAS)])
DENSITY_C = gw_fim.log_density_cros(FIM_PARAM).reshape([len(MCS), len(ETAS)])

# Plot Generation
gw_plt.ripple_waveform(F_SIG, HP, waveform="hp")
gw_plt.ripple_waveform(F_SIG, HC, waveform="hc")
gw_plt.ripple_gradient(F_SIG, HP, HC, param="mc")
gw_plt.ripple_gradient(F_SIG, HP, HC, param="eta")
gw_plt.bilby_noise_psd(F_SIG, F_PSD)
gw_plt.log_fim_contour(MCS, ETAS, DENSITY_P, waveform="hp")
gw_plt.log_fim_contour(MCS, ETAS, DENSITY_C, waveform="hc")
gw_plt.log_fim_param(MCS, DENSITY_P, waveform= "hp",param= "mc")
gw_plt.log_fim_param(ETAS, DENSITY_P, waveform= "hp",param= "eta")
gw_plt.log_fim_param(MCS, DENSITY_C, waveform= "hc",param= "mc")
gw_plt.log_fim_param(ETAS, DENSITY_C, waveform= "hc",param= "eta")

# Flow training
vi_dat.train_flow()

```

[//]: # "==="

Figures

Active figures

- GW150914 simulated waveform with ```ripple.waveforms.IMRPhenomXAS.gen_IMRPhenomXAS_polar```

> ![Test waveform plus](./figures/ripple_hp.png)
> ![Test waveform cros](./figures/ripple_hc.png)

- GW150914 simulated gradient with ```jax.vmap(jax.grad())```

> ![Test gradient wrt chirp mass](./figures/ripple_grad_mc.png)
> ![Test gradient wrt mass ratio](./figures/ripple_grad_eta.png)

- Power Spectral Density of aLIGO from ```bilby```

> ![aLIGO PSD](./figures/bilby_psd.png)

- Log template density

> ![Log template density for waveform plus](./figures/log_fim_contour_hp_1.0_21.0.png)
> ![Log template density for waveform cros](./figures/log_fim_contour_hc_1.0_21.0.png)

```
174
175 - Log template density at different range
176
177 > ![Log template density for waveform plus at lower
    ↪  mc](./figures/log_fim_contour_hp_1.0_2.0.png)
178 > ![Log template density for waveform plus at low mc](./figures/log_fim_contour_hp_1.0_6.0.png)
179 > ![Log template density for waveform plus at high
    ↪  mc](./figures/log_fim_contour_hp_21.0_26.0.png)
180 > ![Log template density for waveform plus at higher
    ↪  mc](./figures/log_fim_contour_hp_21.0_61.0.png)
181
182 - Normalising flow test
183
184 > ![BVM direct](./results/flow_posterior_calc.png)
185 > ![BVM approx](./results/flow_posterior.png)
186
187 - Normalising flow training
188
189 > ![Training loss](./results/flow_loss.png)
190 > ![Training iterations](./results/flow_animation.gif)
191
192 ### Legacy figures
193
194 - GW150914 waveform generated with ```ripplegw```
195
196 > ![Test GW Waveform](./legacy/figures_legacy/fig_01_ripple_waveform.png)
197
198 - GW150914 waveform gradient plot with ```jax.vmap(jax.grad())```
199
200 > ![Test GW Waveform Gradient](./legacy/figures_legacy/fig_02_ripple_waveform_grad.png)
201
202 - PSD aLIGO noise curve with ```bilby```
203
204 > ![Detector PSD](./legacy/figures_legacy/fig_03_bilby_psd.png)
205
206 - Fisher Information Matrix for test GW params
207
208 > ![Test FIM Heat Map](./legacy/figures_legacy/fig_04_fim_heatmap.png)
209
210 - Fisher Information Matrix wrt chirp mass and symmetric mass ratio
211
212 > ![FIM 1-D](./legacy/figures_legacy/fig_05_fim_mc_mr.png)
213
214 - Fisher Information Matrix contour plot
215
216 > ![Density Contour Plot](./legacy/figures_legacy/fig_06_fim_mc_mr_contour.png)
217
218 - Projected metric density contour plot
219
220 > ![Projected Density Contour PLot](./legacy/figures_legacy/fig_06_fim_mc_mr_contour_log10.png)
```

B. Module file: gw_cfg.py

```python
"""
Configuration setup script.
"""
# Library import
import jax
import jax.numpy as jnp
import bilby

# Config setup
# ============================================================================ #
# Frequency - min, max, step
F_MIN, F_MAX, F_DEL = 24.0, 512.0, 0.5
# Chirp mass - min, max, step
MC_MIN, MC_MAX, MC_NUM = 1.000, 21.00, 100
# Mass ratio - min, max, step
ETA_MIN, ETA_MAX, ETA_NUM = 0.050, 0.250, 100
# Base param - mc, eta, s1, s2, dl, tc, phic, theta, phi
PARAM_BASE = jnp.array([28.0956, 0.2471, 0.0, 0.0, 40.0, 0.0, 0.0, 0.0, 0.0])
# Test param for FIM compilation
MC, ETA = 28.0956, 0.2471
PARAM_TEST = jnp.array([MC, ETA, 0.0, 0.0])
# ============================================================================ #

# Frequency array builder


def freq_ripple(data_min: float, data_max: float, data_del: float):
    '''
    Build signal and reference frequency array
    '''
    return jnp.arange(data_min, data_max, data_del), data_min


def freq_fisher(data_min: float, data_max: float, data_del: float):
    '''
    Calculate frequency difference, sampling size, and duration
    '''
    return (
        data_max - data_min,
        (data_max - data_min) / data_del,
        1 / data_del,
    )


def freq_psd(data_samp: float, data_dura: float):
    '''
    Produce bilby based PSD noise array
    '''
    # Get detector
    detector = bilby.gw.detector.get_empty_interferometer("H1")
    # Get sampling freq
    detector.sampling_frequency = data_samp
    # Get dectector duration
    detector.duration = data_dura
    # Return psd as func result
    return detector.power_spectral_density_array[1:]

```

```
58
59  # Theta tuple builder
60
61
62  def theta_ripple(
63      mc_repo: jnp.ndarray,
64      mr_repo: jnp.ndarray,
65      theta: jnp.ndarray,
66  ):
67      '''
68      Create matrix of ripplegw theta arguments
69      '''
70      # Custom concatenater
71      def theta_join(matrix):
72          # Return joined matrix
73          return jnp.concatenate((matrix, theta))
74      # Build mc and mr grid
75      mc_grid, mr_grid = jnp.meshgrid(mc_repo, mr_repo)
76      # Construct (mc, mr) matrix
77      matrix = jnp.stack((mc_grid.flatten(), mr_grid.flatten()), axis=-1)
78      # Return joined matrix
79      return jax.vmap(theta_join)(matrix)
80
81
82  # Generate results
83  # Freq - signal, reference
84  F_SIG, F_REF = freq_ripple(F_MIN, F_MAX, F_DEL)
85  # Freq - difference, sampling, duration
86  F_DIFF, F_SAMP, F_DURA = freq_fisher(F_MIN, F_MAX, F_DEL)
87  # Freq - bilby PSD results
88  F_PSD = freq_psd(F_SAMP, F_DURA)
89  # Chirp mass repo
90  MCS = jnp.linspace(MC_MIN, MC_MAX, MC_NUM)
91  # Mass ratio repo
92  ETAS = jnp.linspace(ETA_MIN, ETA_MAX, ETA_NUM)
93  # Theta matrix result
94  theta_repo = theta_ripple(MCS, ETAS, PARAM_BASE)
```

C. Module file: gw_rpl.py

```python
"""
GW waveform and gradient calculator functions.
"""
# Library import
import os
# Package - jax
import jax
import jax.numpy as jnp
# Package - ripple
from ripple.waveforms import IMRPhenomXAS
# Custom config import
from data.gw_cfg import F_SIG, F_REF, PARAM_BASE, F_PSD, F_DIFF
# XLA GPU resource setup
os.environ['XLA_PYTHON_CLIENT_PREALLOCATE'] = 'false'
jax.config.update("jax_enable_x64", True)

# Ripple - Inner Product Handler


def inner_prod(vec_a: jnp.ndarray, vec_b: jnp.ndarray):
    """
    Noise weighted inner product between vectors a and b
    """
    # Get components
    numerator = jnp.abs(vec_a.conj() * vec_b)
    integrand = numerator / F_PSD
    # Return one side noise weighted inner products
    return 4 * F_DIFF * integrand.sum(axis=-1)


# Ripple - Get waveform_plus -> restricted and normalized


def waveform_plus_restricted(params: jnp.ndarray, freq: jnp.ndarray):
    '''
    Function to return restricted waveform_plus where params are:
    [Mc, eta, t_c, phi_c]
    '''
    # Set complete ripple_theta
    theta = PARAM_BASE.at[0:2].set(params[0:2]).at[5:7].set(params[2:4])
    # Generate plus polarized waveform
    h_plus, _ = IMRPhenomXAS.gen_IMRPhenomXAS_polar(freq, theta, F_REF)
    # Func return
    return h_plus


def waveform_plus_normed(params: jnp.ndarray, freq: jnp.ndarray):
    '''
    Produce waveform normalization for restricted waveoform_plus
    '''
    # Get restricted waveform
    waveform = waveform_plus_restricted(params, freq)
    # Calculate normalization factor
    norm_factor_squared = inner_prod(waveform, waveform)
    # Return normalized waveform
    return waveform / jnp.sqrt(norm_factor_squared)

```

```python

# Ripple - Get waveform_cros -> restricted and normalized


def waveform_cros_restricted(params: jnp.ndarray, freq: jnp.ndarray):
    '''
    Function to return restricted waveform_cros where params are:
    [Mc, eta, t_c, phi_c]
    '''
    # Set complete ripple_theta
    theta = PARAM_BASE.at[0:2].set(params[0:2]).at[5:7].set(params[2:4])
    # Generate cross polarized waveform
    _, h_cros = IMRPhenomXAS.gen_IMRPhenomXAS_polar(freq, theta, F_REF)
    # Func return
    return h_cros


def waveform_cros_normed(params: jnp.ndarray, freq: jnp.ndarray):
    '''
    Produce waveform normalization for restricted waveoform_cros
    '''
    # Get restricted waveform
    waveform = waveform_cros_restricted(params, freq)
    # Calculate normalization factor
    norm_factor_squared = inner_prod(waveform, waveform)
    # Return normalized waveform
    return waveform / jnp.sqrt(norm_factor_squared)


# Ripple - Gradient Calculator


def gradient_plus(theta: jnp.ndarray):
    '''
    Map normalized waveform_plus gradients to signal frequency
    '''
    # Assemble params -- FutureWarning: dtype complex128 -> float64 imcompatible
    params = jnp.array(theta, dtype=jnp.complex128)
    # Return gradiant func mapped to signal frequency array
    return jax.vmap(
        jax.grad(waveform_plus_normed, holomorphic=True),
        in_axes=(None, 0),
    )(params, F_SIG)


def gradient_cros(theta: jnp.ndarray):
    '''
    Map normalized waveform_cros gradients to signal frequency
    '''
    # Assemble params
    # FutureWarning: dtype complex128 -> float64 imcompatible
    params = jnp.array(theta, dtype=jnp.complex128)
    # Return gradiant func mapped to signal frequency array
    return jax.vmap(
        jax.grad(waveform_cros_normed, holomorphic=True),
        in_axes=(None, 0),
    )(params, F_SIG)
```

D. Module file: gw_fim.py

```python
"""
Fisher Information Matrix calculator functions.
"""
# Library import
import os
# Package - jax
import jax
import jax.numpy as jnp
# Other imports
from tqdm import trange
# Custom config import
from data import gw_rpl
# XLA GPU resource setup
os.environ['XLA_PYTHON_CLIENT_PREALLOCATE'] = 'false'
jax.config.update("jax_enable_x64", True)

# FIM - Parameter assembler


def fim_param_build(mcs: jnp.ndarray, etas: jnp.ndarray):
    """
    Build 4-D FIM_PARAM grid with mc and eta entries:
    [mc, eta, tc, phic]
    With input array shape (n, ) and (n, )
    Yield param array shape (n**2, 4)
    """
    # Set (1, ) shape zero value array
    zeros = jnp.zeros(1)
    # Param array - mc, eta, tc, phic
    param_arr = [mcs, etas, zeros, zeros]
    # Build 4-d mesh with matrix indexing
    nd_param = jnp.meshgrid(*param_arr, indexing='ij')
    # Stack and reshape into (n, 4) shape fim_param array
    fim_param = jnp.stack(nd_param, axis=-1).reshape(-1, len(param_arr))
    # Func return - (n**2, 4) param array
    return fim_param


def fim_param_stack(mcs: jnp.ndarray, etas: jnp.ndarray):
    """
    Build FIM_PARAM with column_stack method to get
    [mc, eta, tc, phic]
    With input array shape (n, ) and (n, )
    Yield param array shape (n, 4)
    """
    # Build tc, phic zeros entry array
    zeros = jnp.zeros_like(mcs)
    # Func return - stacked (n, 4) param array
    return jnp.column_stack((mcs, etas, zeros, zeros))


# FIM mapped


@jax.jit
def map_density_plus(param: jnp.ndarray):
    """
```

```
    Return the vectorized template density function for hp
    Param follows shape (n, 4)
    """
    return jax.vmap(log_sqrt_det_plus)(param)


@jax.jit
def map_density_cros(param: jnp.ndarray):
    """
    Return the vectorized template density function for hc
    Param follows shape (n, 4)
    """
    return jax.vmap(log_sqrt_det_cros)(param)


# @jax.jit
def log_density_plus(param: jnp.ndarray):
    """
    Return the vmap generated log density array for hp based param
    """
    # Local resources
    num_param = param.shape[0]
    batch_size = int(num_param * 0.1)
    num_batch = num_param // batch_size
    # Init
    density_list = []
    # Batching
    for i in range(num_batch):
        # Split batches
        batch_fim_param = param[i * batch_size: (i + 1) * batch_size]
        # Call jax.vmap
        batch_density = map_density_plus(batch_fim_param)
        # Add to results
        density_list.append(batch_density)
    # Concatenate the results from smaller batches
    density = jnp.concatenate(density_list)
    return density


# @jax.jit
def log_density_cros(param: jnp.ndarray):
    """
    Return the vmap generated log density array for hc based param
    """
    # Local resources
    num_param = param.shape[0]
    batch_size = int(num_param * 0.1)
    num_batch = num_param // batch_size
    # Init
    density_list = []
    # Batching
    for i in range(num_batch):
        # Split batches
        batch_fim_param = param[i * batch_size: (i + 1) * batch_size]
        # Call jax.vmap
        batch_density = map_density_cros(batch_fim_param)
        # Add to results
        density_list.append(batch_density)
```

```
    # Concatenate the results from smaller batches
    density = jnp.concatenate(density_list)
    return density


# FIM - Main ==> Batching


@jax.jit
def density_batch_calc(
    data: jnp.ndarray,
    mcs: jnp.ndarray,
    etas: jnp.ndarray,
    batch_size: int = 100,
    waveform: str = "hp",
):
    """
    Calculate metric density values with default batching size 100
    Default at waveform hp results
    Not actively used at the moment
    """
    # Select waveform
    if waveform == 'hp':
        wf_func = log_sqrt_det_plus
    elif waveform == 'hc':
        wf_func = log_sqrt_det_cros
    # Define batch numbers
    num_batch = data.shape[0] // batch_size
    density_list = []
    # Batching
    with trange(data.shape[0], desc="Processing Params") as param_range:
        for i in trange(num_batch):
            # Split batches
            batch_fim_param = data[i * batch_size: (i + 1) * batch_size]
            # Call jax.vmap
            batch_density = jax.vmap(wf_func)(batch_fim_param)
            # Add to results
            density_list.append(batch_density)
            # Update progress bar
            param_range.update(batch_fim_param.shape[0])
    # Concatenate the results from smaller batches
    density = jnp.concatenate(density_list).reshape([len(mcs), len(etas)])
    # Func return
    return density


# FIM - Main ==> log.sqrt.det.FIM


def log_sqrt_det_plus(param: jnp.ndarray):
    """
    Return the log based square root of the determinant of
    Fisher matrix projected onto the mc, eta space
    for hp waveform results
    """
    # Calculation
    #try:
    data_fim = projected_fim_plus(param)
```

```
    #except AssertionError:
    #    data_fim = jnp.nan
    # Func return - log density
    return jnp.log(jnp.sqrt(jnp.linalg.det(data_fim)))


def log_sqrt_det_cros(param: jnp.ndarray):
    """
    Return the log based square root of the determinant of
    Fisher matrix projected onto the mc, eta space
    for hc waveform results
    """
    # Calculation
    #try:
    data_fim = projected_fim_cros(param)
    #except AssertionError:
    #    data_fim = jnp.nan
    # Func return - log density
    return jnp.log(jnp.sqrt(jnp.linalg.det(data_fim)))


# FIM projection sub func


def fim_phic(full_fim: jnp.ndarray, nd_val: int):
    """
    Calculate the conditioned matrix projected onto coalecense phase
    """
    # Equation 16 from Dent & Veitch
    fim_result = jnp.array([
        full_fim[i, j] - full_fim[i, -1] * full_fim[-1, j] / full_fim[-1, -1]
        for i in range(nd_val-1)
        for j in range(nd_val-1)
    ]).reshape([nd_val-1, nd_val-1])
    # Func return
    return fim_result


def fim_tc(gamma: jnp.ndarray, nd_val: int):
    """
    Project the conditional matrix back onto coalecense time
    """
    # Equation 18 Dent & Veitch
    fim_result = jnp.array([
        gamma[i, j] - gamma[i, -1] * gamma[-1, j] / gamma[-1, -1]
        for i in range(nd_val-2)
        for j in range(nd_val-2)
    ]).reshape([nd_val-2, nd_val-2])
    # Func return
    return fim_result


# %%
# FIM - Projected and simple FIM


def projected_fim_plus(params: jnp.ndarray):
    """
```

```
    Return the Fisher matrix projected onto the mc, eta space
    for hp waveform results
    """
    # Get full FIM and dimensions
    full_fim = fim_plus(params)
    nd_val = params.shape[-1]
    # Calculate the conditioned matrix for phase
    gamma = fim_phic(full_fim, nd_val)
    # Calculate the conditioned matrix for time
    metric = fim_tc(gamma, nd_val)
    # Func return
    return metric


def projected_fim_cros(params: jnp.ndarray):
    """
    Return the Fisher matrix projected onto the mc, eta space
    for hc waveform results
    """
    # Get full FIM and dimensions
    full_fim = fim_cros(params)
    nd_val = params.shape[-1]
    # Calculate the conditioned matrix for phase
    gamma = fim_phic(full_fim, nd_val)
    # Calculate the conditioned matrix for time
    metric = fim_tc(gamma, nd_val)
    # Func return
    return metric


# FIM packers


def fim_plus(params: jnp.ndarray):
    """
    Returns the fisher information matrix
    at a general value of mc, eta, tc, phic
    for hp waveform

    Args:
        params (array): [Mc, eta, t_c, phi_c]. Shape 1x4
    """
    # Generate the waveform derivatives
    grads = gw_rpl.gradient_plus(params)
    # Get dimensions
    nd_val = grads.shape[-1]
    # Get FIM result
    fim_result = fim_base(grads, nd_val)
    # Func return
    return fim_result


def fim_cros(params: jnp.ndarray):
    """
    Returns the fisher information matrix
    at a general value of mc, eta, tc, phic
    for hc waveform

```

```
    Args:
        params (array): [Mc, eta, t_c, phi_c]. Shape 1x4
    """
    # Generate the waveform derivatives
    grads = gw_rpl.gradient_cros(params)
    # Get dimensions
    nd_val = grads.shape[-1]
    # Get FIM result
    fim_result = fim_base(grads, nd_val)
    # Func return
    return fim_result


def fim_base(grads: jnp.ndarray, nd_val: int):
    """
    Basic FIM entry packer
    """
    # Get FIM entries from inner products calculations
    entries = {
        (i, j): gw_rpl.inner_prod(grads[:, i], grads[:, j])
        for j in range(nd_val)
        for i in range(j+1)
    }
    # Fill the matrix from the precalculated entries
    fim_result = jnp.array([
        entries[tuple(sorted([i, j]))]
        for j in range(nd_val)
        for i in range(nd_val)
    ]).reshape([nd_val, nd_val])
    # Func return
    return fim_result
```

E. Module file: gw_plt.py

```python
"""
Plotter functions repository.
"""
# Library import
import io
import jax.numpy as jnp
import matplotlib.pyplot as plt
import corner
from PIL import Image
import scienceplots
# Plotter style customization
plt.style.use(['science', 'notebook', 'grid'])

# FIM plots


def log_fim_contour(
    data_x: jnp.ndarray,
    data_y: jnp.ndarray,
    data_z: jnp.ndarray,
    waveform: str = 'hp',
):
    """
    Generate contourf plots for log density wrt mc, eta
    Defaulted at waveform hp results
    """
    # Local plotter resources
    xlabel, ylabel, cblabel = (
        r'Chirp Mass $\mathcal{M} [M_\odot$]',
        r'Symmetric Mass Ratio $\eta$',
        r'$\log$ Template Bank Density',
    )
    mc_min, mc_max = jnp.min(data_x), jnp.max(data_x)
    save_path = f'./figures/log_fim_contour_{waveform}_{mc_min}_{mc_max}.png'
    # Plot init
    fig, ax = plt.subplots(figsize=(8, 6))
    # Plotter
    cs = ax.contourf(
        data_x,
        data_y,
        data_z.T,
        alpha=0.8,
        levels=20,
        cmap='gist_heat',
    )
    # Plot customization
    ax.set(xlabel=xlabel, ylabel=ylabel)
    cb = plt.colorbar(cs, ax=ax)
    cb.ax.set_ylabel(cblabel)
    # Plot admin
    fig.savefig(save_path)


def log_fim_param(
    data_x: jnp.ndarray,
    data_y: jnp.ndarray,
    waveform: str = "hp",
```

```
    param: str = "mc",
):
    """
    1-D param plot for log density wrt param entry
    """
    # Local plotter resources
    xlabel_dict = {
        'mc': r'Chirp Mass $\mathcal{M} [M_\odot$]',
        'eta': r'Symmetric Mass Ratio $\eta$',
    }
    ylabel = r'$\log$ Template Bank Density'
    save_path = f'./figures/log_fim_param_{param}_{waveform}.png'
    # Plot init
    fig, ax = plt.subplots(figsize=(8, 6))
    # Plotter
    plot = [
        ax.scatter(
            data_x,
            data_y[:, i],
            alpha=0.8,
            s=1,
            cmap='gist_heat',
            c=data_y[:, i],
        )
        for i in range(int(data_y.shape[0]))
    ]
    # Plot customization
    ax.set(xlabel=xlabel_dict[param], ylabel=ylabel)
    fig.tight_layout()
    # Plot admin
    fig.savefig(save_path)

# Waveform and gradient


def ripple_waveform(
    data_x: jnp.ndarray,
    data_y: jnp.ndarray,
    waveform: str = 'hp',
):
    """
    Generate plots for ripple generated waveforms
    Defaulted at hp waveform
    """
    # Label plotter resources
    if waveform == "hp":
        label1, label2, xlabel, ylabel = (
            r'$\Re h_+$',
            r'$\Im h_+$',
            r'Freqency $f$ [Hz]',
            r'Signal Strain $h_+$',
        )
    elif waveform == "hc":
        label1, label2, xlabel, ylabel = (
            r'$\Re h_\times$',
            r'$\Im h_\times$',
            r'Freqency $f$ [Hz]',
            r'Signal Strain $h_\times$',
```

```
        )
    save_path = f'./figures/ripple_{waveform}.png'
    # Plot init
    fig, ax = plt.subplots(figsize=(8, 6))
    # Plotter
    ax.plot(data_x, data_y.real, label=label1)
    ax.plot(data_x, data_y.imag, label=label2)
    # Plot customization
    ax.set(xlabel=xlabel, ylabel=ylabel, xscale="log")
    ax.legend()
    fig.tight_layout()
    # Plot admin
    fig.savefig(save_path)


def ripple_gradient(
    data_x: jnp.ndarray,
    data_y1: jnp.ndarray,
    data_y2: jnp.ndarray,
    param: str = 'mc',
):
    """
    Generate plots for gradients of ripple generated waveforms
    Defaulted at gradient wrt mc param
    """
    # Local label dict
    label_dict = {
        'real' : r'$\Re$',
        'imag' : r'$\Im$',
        'hp' : r'$h_+$',
        'hc' : r'$h_\times$',
        'freq' : r'Frequency $f$ [Hz]',
        'grad' : r'Gradient wrt. ',
        'mc': r'Chirp Mass $\tilde{h}_{\mathcal{M}}$',
        'eta': r'Symmetric Mass Ratio $\tilde{h}_{\eta}$',
        's1': r'Spin of $m_1$ $\tilde{h}_{s_1}$',
        's2': r'Spin of $m_2$ $\tilde{h}_{s_2}$',
        'dl': r'Distance $\tilde{h}_{d_L}$',
        'tc': r'Coalescence Time $\tilde{h}_{t_c}$',
        'phic': r'Coalescence Phase $\tilde{h}_{\phi_c}$',
        'theta': r'Inclination Angle $\tilde{h}_{\theta}$',
        'phi': r'Polarization Angle $\tilde{h}_{\phi}$',
    }
    # Local plotter resources
    label1, label2, xlabel, ylabel = (
        f"{label_dict['real']}{label_dict['hp']}",
        f"{label_dict['imag']}{label_dict['hc']}",
        f"{label_dict['freq']}",
        f"{label_dict['grad']}{label_dict[param]}",
    )
    save_path = f'./figures/ripple_{param}.png'
    # Plot init
    fig, ax = plt.subplots(figsize=(8, 6))
    # Plotter
    ax.plot(data_x, data_y1.real, label=label1)
    ax.plot(data_x, data_y2.imag, label=label2)
    # Plot customization
    ax.set(xlabel=xlabel, ylabel=ylabel, xscale="log")
```

```
    ax.legend()
    fig.tight_layout()
    # Plot admin
    fig.savefig(save_path)


# PSD from bilby


def bilby_noise_psd(data_x: jnp.ndarray, data_y: jnp.ndarray):
    """
    Plot the PSD obtained from bilby
    """
    # Local plotter resources
    data_min, color_red, color_blue = (
        0.0,
        '#B30C00',
        '#005398',
    )
    label, xlabel, ylabel,  = (
        r'H1 Power Spectral Density $S(f)$',
        r'Frequency $f$ [Hz]',
        r'GW Strain Noise [Hz^($-1/2$)]',
    )
    save_path = './figures/bilby_psd.png'
    # Plot init
    fig, ax = plt.subplots(figsize=(8, 6))
    # Plotter
    ax.plot(data_x, data_y, label=label, alpha=0.8, lw=2, color=color_red)
    ax.fill_between(data_x, data_y, data_min, alpha=0.6, color=color_blue)
    # Plot customization
    ax.set(xlabel=xlabel, ylabel=ylabel, xscale='log', yscale='log')
    ax.legend()
    fig.tight_layout()
    # Plot admin
    fig.savefig(save_path)


# Flow results gif plotter


def make_gif(data_flow):
    """
    GIF generator for flow results
    """
    # Frame repo init
    frames = []
    # Frame generation
    # for i in range(len(data_flow)):
    for _, flow in enumerate(data_flow):
        # Plot epoch related flow results
        corner.corner(flow)
        # Create frame buffer
        img_buf = io.BytesIO()
        # Save frames to buffer
        plt.savefig(img_buf, format='png')
        # Re-init
        plt.close()
```

```
            # Add to frame repo
            image = Image.open(img_buf)
            frames.append(image)
        # Get first frame
        frame_one = frames[0]
        # Save fig
        frame_one.save(
            #f'./results/{RUN_NAME}_animation.gif',
            './results/flow_animation.gif',
            format="GIF",
            append_images=frames,
            save_all=True,
            duration=100,
            loop=0,
        )
        # Terminate buffer
        img_buf.close()
```

F. Module file: vi_cfg.py

```python
"""
VI config file.
"""
# Library import
import haiku as hk
from data import vi_cls

# Target distribution. Bivariate von Mises distribution on a 2-Torus.
LOC = [0.0, 0.0]
CONCENTRATION = [4.0, 4.0]
CORRELATION = 0.0
# Target density params tc, phic
PARAM_RIPPLE = [0.0, 0.0]

# Flow parameters
NUM_PARAMS = 2
NUM_FLOW_LAYERS = 2
HIDDEN_SIZE = 8
NUM_MLP_LAYERS = 2
NUM_BINS = 4

# Perform variational inference
TOTAL_EPOCHS = 3000 #reduce this for testing purpose, original val = 10000
NUM_SAMPLES = 1000 #1000
LEARNING_RATE = 0.001 #0.001

# Other cfg
PRNG_SEQ = hk.PRNGSequence(42)

# Target distribution selector
DIST_BVM = vi_cls.BivariateVonMises(LOC, CONCENTRATION, CORRELATION)
DIST_GW = vi_cls.TemplateDensity(PARAM_RIPPLE)
```

G. Module file: vi_cls.py

```python
"""
VI target distribution classes repo.
"""
# Library import
import jax.numpy as jnp
from data import gw_fim

# Dist - BVM distribution


class BivariateVonMises:
    """
    Original BVM class
    """
    def __init__(self, loc, concentration, correlation):
        self.data_mu, self.data_nu = loc
        self.data_k1, self.data_k2 = concentration
        self.data_k3 = correlation


    # Target dist, need to switch to working dist of fim density
    def log_prob(self, data_x):
        '''
        Get the log probability distribution
        '''
        # 2-D parameters
        phi, psi = data_x.T
        phi = 2*jnp.pi*phi
        psi = 2*jnp.pi*psi
        # Get result
        result = (
            self.data_k1*jnp.cos(phi - self.data_mu)
            + self.data_k2*jnp.cos(psi - self.data_nu)
            - self.data_k3*jnp.cos(phi - self.data_mu - psi + self.data_nu)
        )
        # Func return
        return result


    def prob(self, data_x):
        """
        Get probability distribution
        """
        # Func return - examine log_prob(input), the input may be inverted
        return jnp.exp(self.log_prob(data_x))


# Dist - GW Metric Density


class TemplateDensity:
    """
    GW template density class for hp based results
    """
    def __init__(self, param_ripple):
        self.data_tc, self.data_phic = param_ripple

```

```
58
59      def log_prob(self, data_x):
60          '''
61          Get the log template density
62          '''
63          # Local assignment, 2-d param
64          # data_x.shape (n, 2)
65          data_mc, data_eta = data_x.T
66          data_mc = data_mc + 1.0 # 1.0 - 2.0
67          data_eta = data_eta * 0.24 + 0.01 # 0.05 - 0.25
68          # Param build with shape (n, 4)
69          param = gw_fim.fim_param_stack(data_mc, data_eta)
70          # Get results with shape (n, )
71          result = gw_fim.map_density_plus(param)
72          # Func return
73          return result
74
75
76      def prob(self, data_x):
77          """
78          Get probability distribution
79          """
80          # Func return - examine log_prob(input), the input may be inverted
81          return jnp.exp(self.log_prob(data_x))
```

H. Module file: vi_dat.py

```python
"""
VI functions and flow model.
"""
# Library import
from typing import Any, Sequence, Tuple
import haiku as hk
# Package - jax, numpy
import jax
import jax.numpy as jnp
import numpy as np
import optax
import distrax
from tqdm import trange
# Other imports
from data import vi_plt
from data.vi_cfg import (
    NUM_PARAMS,
    NUM_FLOW_LAYERS,
    HIDDEN_SIZE,
    NUM_MLP_LAYERS,
    NUM_BINS,
    NUM_SAMPLES,
    LEARNING_RATE,
    TOTAL_EPOCHS,
    PRNG_SEQ,
    DIST_BVM,
    #DIST_GW,
)
# Aliasing
PRNGKey = jnp.ndarray
OptState = Any


# Get target distribution from config
DIST = DIST_BVM

# Other configs import
OPTIMISER = optax.adam(LEARNING_RATE)
key = next(PRNG_SEQ)

# Flow training function


def train_flow():
    """
    Preliminary flow training script
    Generate and save training loss
    Plot training loss
    Plot approximated posterior
    """
    # Local init
    loss = {"train": [], "val": []}
    ldict = {"loss": 0}
    losses = []
    flows = []

    data_param = sample_and_log_prob.init(key, prng_key=key, data_n=NUM_SAMPLES)
```

```
    data_opt_state = OPTIMISER.init(data_param)

    # Start training print
    print()
    print("Training: Initiated")
    print("=" * 30)

    # Training
    with trange(TOTAL_EPOCHS) as tepochs:
        for epoch in tepochs:
            data_prng_key = next(PRNG_SEQ)
            loss = loss_fn(data_param, data_prng_key, NUM_SAMPLES)
            ldict['loss'] = f'{loss:.2f}'
            losses.append(loss)
            tepochs.set_postfix(ldict, refresh=True)
            # Problematic grad(loss_fn)
            data_param, data_opt_state = update(data_param, data_prng_key, data_opt_state)
            # Results
            if epoch%100 == 0:
                x_gen, log_prob_gen = sample_and_log_prob.apply(
                    data_param,
                    next(PRNG_SEQ),
                    10*NUM_SAMPLES,
                )
                samples = np.array(x_gen)
                flows.append(samples)
                # Print results
                print(f'At epoch: {epoch}, with loss: {loss}')

    # Print if complete
    print("Training accomplished.")
    print("=" * 30)
    print()

    # Save plot of the final posterior
    x_gen, log_prob_gen = sample_and_log_prob.apply(
        data_param,
        next(PRNG_SEQ),
        100*NUM_SAMPLES,
    )
    # Save plot of the loss
    vi_plt.flow_posterior(x_gen)
    vi_plt.training_loss(losses)
    # Plot animation of the flows
    vi_plt.make_gif(flows)

    # Save loss array
    file_loss = open('./results/flow_loss.npy', 'wb')
    np.save(file_loss, np.array(losses))
    file_loss.close()


# Flow model


def make_conditioner(
    event_shape: Sequence[int],
    hidden_sizes: Sequence[int],
```

```
    num_bijector_params: int
    ) -> hk.Sequential:
    """
    Creates a conditioner, a Neural Network (parameters of the spline)
    """
    return hk.Sequential([
        hk.Flatten(preserve_dims=-len(event_shape)),
        hk.nets.MLP(hidden_sizes, activate_final=True),
        # We initialize this linear layer to zero so that the flow is initialized
        # to the identity function.
        hk.Linear(
            np.prod(event_shape) * num_bijector_params,
            w_init=jnp.zeros,
            b_init=jnp.zeros),
        hk.Reshape(tuple(event_shape) + (num_bijector_params,), preserve_dims=-1),
    ])


def make_flow_model(
    event_shape: Sequence[int],
    num_layers: int = 4,
    hidden_sizes: Sequence[int] = [250, 250],
    num_bins: int = 4,
    ) -> distrax.Transformed:
    """
    Creates the normalizing flow model
    """
    # Alternating binary mask.
    mask = np.arange(0, np.prod(event_shape)) % 2
    mask = np.reshape(mask, event_shape)
    mask = mask.astype(bool)
    # Param range definer
    range_min, range_max = 0.0, 1.0 #2*jnp.pi

    # Bijector
    def bijector_fn(params: jnp.ndarray):
        """
        Bijector
        """
        return distrax.RationalQuadraticSpline(
            # Regular spline
            # This defines the domain of the flow parameters
            params, range_min=0.0, range_max= 1.0 #2*jnp.pi
        )


    # Number of parameters for the rational-quadratic spline:
    # - `num_bins` bin widths
    # - `num_bins` bin heights
    # - `num_bins + 1` knot slopes
    # for a total of `3 * num_bins + 1` parameters.
    num_bijector_params = 3 * num_bins + 1

    layers = []
    for _ in range(num_layers):
        layer = distrax.MaskedCoupling(
            mask=mask,
            bijector=bijector_fn,
```

```
            conditioner=make_conditioner(
                event_shape,
                hidden_sizes,
                num_bijector_params,
            )
        )
        layers.append(layer)
        # Flip the mask after each layer.
        mask = jnp.logical_not(mask)

    # We invert the flow so that the `forward` method is called with `log_prob`.
    #bijective transformation from base (normal) to parameter space
    flow = distrax.Inverse(distrax.Chain(layers))
    base_distribution = distrax.Independent(
        #distrax.Uniform(low=jnp.ones(event_shape)*-1, high=jnp.ones(event_shape)*1),
        distrax.Uniform(low=jnp.ones(event_shape)*range_min,
        ↪  high=jnp.ones(event_shape)*range_max),
        #distrax.Normal(loc=jnp.zeros(event_shape), scale=jnp.ones(event_shape)),
        reinterpreted_batch_ndims=len(event_shape)
    )

    return distrax.Transformed(base_distribution, flow)


@hk.without_apply_rng
@hk.transform
def sample_and_log_prob(prng_key: PRNGKey, data_n: int) -> Tuple[Any, jnp.ndarray]:
    """
    Generates the sample parameter entries
    and obtain log prob from the sample param entries
    """
    # Shape
    event_shape=(NUM_PARAMS,)
    # Model
    model = make_flow_model(
        event_shape=event_shape,
        num_layers=NUM_FLOW_LAYERS,
        hidden_sizes=[HIDDEN_SIZE] * NUM_MLP_LAYERS,
        num_bins=NUM_BINS,
    )
    # Func return
    return model.sample_and_log_prob(seed=prng_key, sample_shape=(data_n,))


@hk.without_apply_rng
@hk.transform
def flow_prob(data_x: jnp.ndarray) -> jnp.ndarray:
    """
    Gets the prob values from the sample param entries
    """
    # Get shape
    event_shape=(NUM_PARAMS,)
    # Model
    model = make_flow_model(
        event_shape=event_shape,
        num_layers=NUM_FLOW_LAYERS,
        hidden_sizes=[HIDDEN_SIZE] * NUM_MLP_LAYERS,
        num_bins=NUM_BINS,
```

```
    )
    # Func return
    return model.prob(data_x)


def loss_fn(params: hk.Params, prng_key: PRNGKey, data_n: int) -> jnp.ndarray:
    """
    Calculate the expected value of Kullback-Leibler (KL) divergence
    """
    # Local calculation resources
    x_flow, log_q = sample_and_log_prob.apply(params, prng_key, data_n)
    log_p = DIST.log_prob(x_flow)
    # Get the KL divergence as loss
    data_loss = jnp.mean(log_q - log_p)
    # Func return
    return data_loss


@jax.jit
def update(
    params: hk.Params,
    prng_key: PRNGKey,
    opt_state: OptState,
    ) -> Tuple[hk.Params, OptState]:
    """
    Single SGD update step
    """
    grads = jax.grad(loss_fn)(params, prng_key, NUM_SAMPLES)
    updates, new_opt_state = OPTIMISER.update(grads, opt_state)
    new_params = optax.apply_updates(params, updates)
    # Func return
    return new_params, new_opt_state
```

I. Module file: vi_plt.py

```python
"""
Plotter functions repository for VI related tasks.
"""
# Library import
import io
import matplotlib.pyplot as plt
import corner
import numpy as np
from PIL import Image


def make_gif(data_flow):
    """
    GIF generator for flow results
    """
    # Frame repo init
    frames = []
    # Frame generation
    # for i in range(len(data_flow)):
    for _, flow in enumerate(data_flow):
        # Plot epoch related flow results
        corner.corner(flow)
        # Create frame buffer
        img_buf = io.BytesIO()
        # Save frames to buffer
        plt.savefig(img_buf, format='png')
        # Re-init
        plt.close()
        # Add to frame repo
        image = Image.open(img_buf)
        frames.append(image)
    # Get first frame
    frame_one = frames[0]
    # Save fig
    frame_one.save(
        #f'./results/{RUN_NAME}_animation.gif',
        './results/flow_animation.gif',
        format="GIF",
        append_images=frames,
        save_all=True,
        duration=100,
        loop=0,
    )
    # Terminate buffer
    img_buf.close()


def flow_posterior(x_gen):
    """
    Generate posterior distribution approximated by NF
    """
    corner.corner(
        np.array(x_gen),
        labels=[r'$\psi$', r'$\phi$'],
        plot_density=True,
        plot_datapoints=True,
    )
```

```
58      # plt.savefig(f'./results/{RUN_NAME}_posterior.png')
59      plt.savefig('./results/flow_posterior.png')
60      plt.close()
61
62
63  def training_loss(losses):
64      """
65      Plot training loss
66      """
67      plt.plot(losses, lw='2', alpha=0.8, color='black')
68      plt.xlabel("Iteration")
69      plt.ylabel("Loss")
70      # plt.savefig(f'./results/{RUN_NAME}_loss.png')
71      plt.savefig('./results/flow_loss.png')
72      plt.close()
```

J. Script file: main.py

```python
"""
This is the master script for MSc project.

Created on Thu August 03 2023
"""
# Library import
# Set XLA resource allocation
import os
# Use jax and persistent cache
from jax.experimental.compilation_cache import compilation_cache as cc
# Custom packages
from data import gw_fim, gw_plt, gw_rpl, vi_dat
from data.gw_cfg import MCS, ETAS, PARAM_TEST, F_SIG, F_PSD
# Setup
os.environ['XLA_PYTHON_CLIENT_PREALLOCATE'] = 'false'
cc.initialize_cache("./data/__jaxcache__")

# First compilation test for sub modules
# Wavefor generation
HP = gw_rpl.waveform_plus_restricted(PARAM_TEST, F_SIG)
HC = gw_rpl.waveform_cros_restricted(PARAM_TEST, F_SIG)
# Gradient calculation
GP = gw_rpl.gradient_plus(PARAM_TEST)
GC = gw_rpl.gradient_cros(PARAM_TEST)
# FIM test statistics calculation
DETP = gw_fim.log_sqrt_det_plus(PARAM_TEST)
DETC = gw_fim.log_sqrt_det_cros(PARAM_TEST)
# First compilation - results checker
print(f"Test waveform HP.shape:{HP.shape} hc.shape:{HC.shape}")
print(f"Test gradient gp.shape:{GP.shape} gc.shape:{GC.shape}")
print(f"Test log density detp:{DETP:.4g} detc:{DETC:.4g}")

# FIM density calc params
FIM_PARAM = gw_fim.fim_param_build(MCS, ETAS)
print(f"fim_param.shape:{FIM_PARAM.shape}")

# New compilation for vectorized operaions
DENSITY_P = gw_fim.log_density_plus(FIM_PARAM).reshape([len(MCS), len(ETAS)])
DENSITY_C = gw_fim.log_density_cros(FIM_PARAM).reshape([len(MCS), len(ETAS)])

# Plot Generation
gw_plt.ripple_waveform(F_SIG, HP, waveform="hp")
gw_plt.ripple_waveform(F_SIG, HC, waveform="hc")
gw_plt.ripple_gradient(F_SIG, HP, HC, param="mc")
gw_plt.ripple_gradient(F_SIG, HP, HC, param="eta")
gw_plt.bilby_noise_psd(F_SIG, F_PSD)
gw_plt.log_fim_contour(MCS, ETAS, DENSITY_P, waveform="hp")
gw_plt.log_fim_contour(MCS, ETAS, DENSITY_C, waveform="hc")
gw_plt.log_fim_param(MCS, DENSITY_P, waveform= "hp",param= "mc")
gw_plt.log_fim_param(ETAS, DENSITY_P, waveform= "hp",param= "eta")
gw_plt.log_fim_param(MCS, DENSITY_C, waveform= "hc",param= "mc")
gw_plt.log_fim_param(ETAS, DENSITY_C, waveform= "hc",param= "eta")

# Flow training
vi_dat.train_flow()
```