SUPPLEMENTARY INFORMATION
"Smooth Normalizing Flows"

SI-1 Proofs

SI-1.1 Smooth Bump Functions

We first show that the smooth ramp as defined in Sec. 4 is indeed as smooth as promised.

Lemma 1. *Let $\alpha > 0$ and $\beta \geq 1$. The function given by*

$$\rho(x) := \begin{cases} \exp\left(-\frac{1}{\alpha \cdot x^\beta}\right) & x > 0 \\ 0 & \text{o.w.} \end{cases} \tag{12}$$

is smooth. Furthermore, for all $n \in \mathbb{N}$ we have $\lim_{x \to 0} \partial_x^{(n)} \rho(x) \to 0$.

Proof. We follow classic text-book examples and give the proof for completeness. First note that the cases for $x > 0$ and $x < 0$ are trivial. We thus have to consider $x = 0$. We see that all left-sided derivatives vanish at $x = 0$. Thus, it is sufficient to consider only right-sided derivatives. For $x > 0$ we have

$$\partial_x \rho(x) = \alpha\beta \cdot x^{-(\beta+1)} \rho(x). \tag{13}$$

By induction, we can see that the n-th derivative is given by

$$\partial_x^n \rho(x) = \mathcal{P}\left(x^{-1}\right) \rho(x), \tag{14}$$

where $\mathcal{P}\left(x^{-1}\right)$ is some polynomial in x^{-1}. For any polynomial we have

$$\lim_{y \to \infty} \frac{\mathcal{P}(y)}{\exp\left(\frac{y^\beta}{\alpha}\right)} = 0. \tag{15}$$

Thus, we end up with

$$\lim_{x \to 0} \partial_x^n \rho(x) = 0. \tag{16}$$

□

Now with the following theorem, we see that the smooth bump functions satisfy all necessary conditions.

Theorem 1. *Let $\rho \colon [0, 1] \to [0, 1]$ be a n-times differentiable and strictly monotonously increasing function with $\lim_{x \to 0} \rho(x) = 0$ and $\lim_{x \to 1} \rho(x) = 1$. Then the function*

$$\sigma[\rho](x) := \frac{\rho(x)}{\rho(x) + \rho(1 - x)} \tag{17}$$

is n-times differentiable, strictly monotonously increasing and satisfies

$$\sigma[\rho](0) = 0, \tag{18}$$

$$\sigma[\rho](1) = 1. \tag{19}$$

If furthermore, $\lim_{x \to 0} \partial_x^k \rho(x) = 0$ for all $k \leq n$ we also have

$$\lim_{x \to 0} \partial_x^n \sigma[\rho](x) = \lim_{x \to 1} \partial_x^n \sigma[\rho](x) = 0. \tag{20}$$

Proof. We quickly see that

$$\lim_{x \to 0} \sigma[\rho](x) = \lim_{x \to 0} \frac{\rho(x)}{\rho(x) + \rho(1 - x)} \tag{21}$$

$$= \frac{\lim_{x \to 0} \rho(x)}{\lim_{x \to 0} \rho(x) + \lim_{x \to 0} \rho(1 - x)} \tag{22}$$

$$= \frac{0}{0 + 1} = 0. \tag{23}$$

Analogously, we can show $\lim_{x\to 1}\sigma[\rho](x) = 1$. For the first derivative we compute

$$\partial_x \sigma[\rho](x) = \frac{\partial_x \rho(x)}{\rho(x)+\rho(1-x)} - \frac{\rho(x)(\partial_x\rho(x) - \partial_x\rho(1-x))}{(\rho(x)+\rho(1-x))^2} \tag{24}$$

$$= \frac{(\partial_x\rho(x))\rho(1-x) + (\partial_x\rho(1-x))\rho(x)}{(\rho(x)+\rho(1-x))^2}, \tag{25}$$

from which we see that $\sigma[\rho]$ is strictly monotonously increasing within $(0,1)$ and thus on all of $[0,1]$. As the denominator does not vanish in all of $[0,1]$ and ρ is k-times differentiable, we can conclude that $\sigma[\rho]$ is k-times differentiable as well.

Now let $\lim_{x\to 0}\partial_x^k\rho(x) = 0$ for all $k < n$. We first note that

$$C_k := \lim_{x\to 0}\partial_x^k(\rho(x)+\rho(1-x)) \tag{26}$$

$$= \lim_{x\to 0}\partial_x^k\rho(x) + (-1)^k\partial_x^n\rho(1-x) \tag{27}$$

$$= (-1)^k \lim_{x\to 1}\partial_x^k\rho(x), \tag{28}$$

exists.

Then using the following recursive formula for the n-th derivative of the quotient of two functions given by Xenophontos [56]

$$\partial_x^n \frac{u(x)}{v(x)} = \frac{1}{v(x)}\left[\partial_x^n u(x) - n!\sum_{j=1}^{n}\frac{\partial_x^{n+1-j}v(x)\cdot\partial_x^{j-1}\frac{u(x)}{v(x)}}{(n+1-j)!(j-1)!}\right] \tag{29}$$

and setting $u(x) = \rho(x), v(x) = \rho(x)+\rho(1-x)$, we obtain

$$\lim_{x\to 0}\partial_x^n\sigma[\rho](x) = \lim_{x\to 0}\left(\frac{1}{\rho(x)+\rho(1-x)}\right. \tag{30}$$

$$\left.\cdot\left[\partial_x^n\rho(x) - n!\sum_{j=1}^{n}\frac{\partial_x^{n+1-j}(\rho(x)+\rho(1-x))\cdot\partial_x^{j-1}\sigma[\rho](x)}{(n+1-j)!(j-1)!}\right]\right). \tag{31}$$

We will prove the rest of the theorem by induction. Assume that

$$\lim_{x\to 0}\partial_x^k\sigma[\rho](x) = 0 \tag{32}$$

for all $k < n$. As $\rho(x)+\rho(1-x) > 0$ for all $x\in[0,1]$ and all limits exist we obtain

$$\lim_{x\to 0}\partial_x^n\sigma[\rho](x) = \frac{1}{1}\left[0 - n!\sum_{j=1}^{n}\frac{C_{n+1-j}}{(n+1-j)!(j-1)!}\cdot\lim_{x\to 0}\partial_x^{j-1}\sigma[\rho](x)\right] \tag{33}$$

$$= \frac{1}{1}\left[0 - n!\sum_{j=1}^{n}\frac{C_{n+1-j}}{(n+1-j)!(j-1)!}\cdot 0\right] = 0, \tag{34}$$

$$\tag{35}$$

which proves the induction step. For the base case we evaluate (25) at $x = 0$ which completes the proof. To show that also $\lim_{x\to 1}\partial_x^n\sigma[\rho](x) = 0$ we use that

$$\sigma[\rho](x) = \frac{\rho(x)}{\rho(1-x)+\rho(x)} \tag{36}$$

$$= 1 - \frac{\rho(1-x)}{\rho(1-x)+\rho(x)} \tag{37}$$

$$= 1 - \sigma[\rho](1-x) \tag{38}$$

$$\tag{39}$$

from which we have for $n > 0$

$$\lim_{x\to 1} \partial_x^n \sigma[\rho](x) = \lim_{x\to 0} \partial_x^n \sigma[\rho](1-x) \tag{40}$$
$$= \lim_{x\to 0} \partial_x^n (1 - \sigma[\rho](x)) \tag{41}$$
$$= -\lim_{x\to 0} \partial_x^n \sigma[\rho](x) \tag{42}$$
$$= 0. \tag{43}$$

□

SI-1.2 Circular Wrapping

Here we explain, how the former construction of $\sigma[\rho]$ can be used to define smooth bijections on the circle (and the hypertorus). We first explain the general recipe which works for any compactly supported smooth transformation and then how it is instantiated for the functions as defined in Sec. 4.

General wrapping construction for smooth compact bump functions Let p a smooth density with support on $[a, b]$, such that $b-a \leq 1$ and $[a, b]\cap[0, 1] \neq \emptyset$. Let furthermore $\partial_x^n p(a) = \partial_x^n p(b) = 0$ for all n. Then we can define a smooth density $\hat{p}$ on $[0, 1]$ using the following cases:

- For $[a, b] \subset [0, 1]$ set

$$\hat{p}(x) = \begin{cases} p(x) & x \in [a, b] \\ 0 & \text{o.w.} \end{cases} \tag{44}$$

- For $a < 0$ set

$$\hat{p}(x) = \begin{cases} p(x) & x \in [0, b] \\ p(x-1) & x \in [1+a, 1] \\ 0 & \text{o.w.} \end{cases} \tag{45}$$

- For $b > 1$ set

$$\hat{p}(x) = \begin{cases} p(x) & x \in [a, 1] \\ p(1+x) & x \in [0, b-1] \\ 0 & \text{o.w.} \end{cases} \tag{46}$$

It is easy to see that $\partial_x^n \hat{p}(0) = \partial_x^n \hat{p}(1)$ for all n. For some $c > 0$ we can now define the smooth non-vanishing density $\bar{p}$ on $[0, 1]$ as

$$\bar{p}(x) = (1-c)\hat{p}(x) + c. \tag{47}$$

Then the function

$$f(x) = \int_0^x \bar{p}(z)\, dz. \tag{48}$$

is a smooth bijection on $[0, 1]$ with $\partial_x^n f(0) = \partial_x^n f(1)$. By this, we obtain a smooth bijection on S^1 with periodic derivative $\bar{p}$.

Instantiation for the smooth bump functions of Sec 4 As discussed in Sec. 4 we can choose $b \in [0, 1]$ and $a \geq 1$ and set

$$g(x) = \sigma[\rho]\left(a(x-b) + \tfrac{1}{2}\right) \tag{49}$$

to obtain the smooth $[b-\frac{1}{2a}, b+\frac{1}{2a}]$-supported bump function $\partial_x g$. This satisfies the assumptions on p of the former paragraph. Thus, we can set $p := \partial_x g$ which gives us the desired smooth transformation $f\colon S^1 \to S^1$. Note, that due to the piece-wise construction of $\bar{p}$, we can evaluate the integral (48) in closed form.

SI-1.3 Differentiation through Blackbox Root-Finding

Gradient relations Here we derive the gradient relations as discussed in Sec. 5. This is a generalization of the usual inverse function theorem in 1D to the case of scalar bijections conditioned on some parameter $\boldsymbol{\theta}$. For reasons of brevity in the main text and precision in the proof, we used a different notation in Sec. 5 compared to the one we will use here. Specifically, we will denote the forward transform as $\alpha(\cdot;\boldsymbol{\theta})$ while we denote the inverse transform as $\beta(\cdot;\boldsymbol{\theta})$ for some parameters $\boldsymbol{\theta} \in \mathbb{R}^d$.

Theorem 2. *Let $\Omega \subset \mathbb{R}$ open and*

$$\alpha\colon \Omega \times \mathbb{R}^d \to \Omega, \qquad \beta\colon \Omega \times \mathbb{R}^d \to \Omega$$

be $\mathcal{C}^2$ functions such that for all $\boldsymbol{\theta} \in \mathbb{R}^d$, $x \in \Omega$ and $y := \alpha(x, \boldsymbol{\theta})$

$$\alpha(\beta(y;\boldsymbol{\theta});\boldsymbol{\theta}) = y \qquad \beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = x \tag{50}$$

and

$$\partial_x \alpha(x;\boldsymbol{\theta}) > 0 \qquad \partial_y \beta(y;\boldsymbol{\theta}) > 0. \tag{51}$$

Define further

$$g_\alpha(x,\boldsymbol{\theta}) := \log\left|\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right|, \quad g_\beta(y,\boldsymbol{u}) := \log\left|\frac{\partial\beta(y;\boldsymbol{u})}{\partial y}\right|. \tag{52}$$

Then we have the following gradient relations:

$$\frac{\partial\beta(y;\boldsymbol{\theta})}{\partial y} = \left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1} \tag{53}$$

$$\frac{\partial\beta(y;\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{54}$$

$$\frac{\partial g_\beta(y;\boldsymbol{\theta})}{\partial y} = -\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1}\frac{\partial g_\alpha(x;\boldsymbol{\theta})}{\partial x} \tag{55}$$

$$\frac{\partial g_\beta(y;\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \left(\frac{\partial\alpha(y;\boldsymbol{\theta})}{\partial x}\right)^{-1}\left(\frac{\partial g_\alpha(x;\boldsymbol{\theta})}{\partial x}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} - \frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial x}\right) \tag{56}$$

Proof. We have

$$\frac{d}{dx}\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x} = \frac{dx}{dx} = 1. \tag{57}$$

and thus

$$\frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y} = \left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1}. \tag{58}$$

Now

$$\frac{d}{d\boldsymbol{\theta}}\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} + \frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} = \frac{\partial x}{\partial\boldsymbol{\theta}} = 0. \tag{59}$$

which gives

$$\frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} = \frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \tag{60}$$

$$= \left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \tag{61}$$

Combining

$$\frac{d}{dx}\frac{d}{dx}\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \frac{d}{dx}\frac{dx}{dx} = 0 \tag{62}$$

and

$$\frac{d}{dx}\frac{d}{dx}\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \frac{d}{dx}\frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x} \tag{63}$$

$$= \frac{\partial^2\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y\partial y}\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^2 + \frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial x\partial x} \tag{64}$$

we obtain

$$\frac{\partial^2\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y\partial y} = -\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-2}\frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial x\partial x} \tag{65}$$

$$= -\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-3}\frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial x\partial x} \tag{66}$$

$$= -\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-2}\frac{\partial}{\partial x}\log\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right) \tag{67}$$

$$= -\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-2}\frac{\partial g_\alpha(x;\boldsymbol{\theta})}{\partial x} \tag{68}$$

and thus

$$\frac{\partial}{\partial y}g_\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \left(\frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\right)^{-1}\frac{\partial^2\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y\partial y} \tag{69}$$

$$= -\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1}\frac{\partial g_\alpha(x;\boldsymbol{\theta})}{\partial x} \tag{70}$$

Finally, combining

$$\frac{d}{d\boldsymbol{\theta}}\frac{d}{dx}\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\frac{dx}{dx} = 0 \tag{71}$$

and

$$\frac{d}{d\boldsymbol{\theta}}\frac{d}{dx}\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x} \tag{72}$$

$$\begin{aligned} &= \frac{\partial^2\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x} \\ &\quad + \frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial x} \\ &\quad + \frac{\partial^2\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x} \end{aligned} \tag{73}$$

gives us

$$\frac{\partial^2\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial y} = -\frac{\partial^2\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y\partial y}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \tag{74}$$

$$\begin{aligned} &- \frac{\partial\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial x}\left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1} \\ &= \left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-2}\frac{\partial g_\alpha(x;\boldsymbol{\theta})}{\partial x}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \end{aligned} \tag{75}$$

$$\begin{aligned} &- \left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-2}\frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial x} \\ &= \left(\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-2}\left(\frac{\partial g_\alpha(x;\boldsymbol{\theta})}{\partial x}\frac{\partial\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} - \frac{\partial^2\alpha(x;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial x}\right) \end{aligned} \tag{76}$$

And thus we get

$$\frac{\partial}{\partial \boldsymbol{\theta}} g_\beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta}) = \left(\frac{\partial \beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial y}\right)^{-1} \frac{\partial^2 \beta(\alpha(x;\boldsymbol{\theta});\boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial y} \tag{77}$$

$$= \left(\frac{\partial \alpha(x;\boldsymbol{\theta})}{\partial x}\right)^{-1} \left(\frac{\partial g_\alpha(x;\boldsymbol{\theta})}{\partial x} \frac{\partial \alpha(x;\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} - \frac{\partial^2 \alpha(x;\boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial x}\right) \tag{78}$$

□

Use in component-wise transformations Now we discuss component-wise transformations, i.e. *transformers* of coupling layers. [2] Let the multivariate transformations $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ factorize as

$$\boldsymbol{\alpha}(\boldsymbol{x},\boldsymbol{\theta})_i = \alpha_i(x_i,\boldsymbol{\theta}) \tag{79}$$

$$\boldsymbol{\beta}(\boldsymbol{y},\boldsymbol{\theta})_i = \beta_i(y_i,\boldsymbol{\theta}) \tag{80}$$

for some component bijections α_i, β_i, which is equivalent to $\partial_{\boldsymbol{x}}\boldsymbol{\alpha}$ and $\partial_{\boldsymbol{y}}\boldsymbol{\beta}$ being diagonal.

First, note that all component-wise transformations within automatic differentation graphs (e.g. component-wise applied `relu`, `exp` or `log` operations) only require the computation of component-wise derivatives in order to compute vector-jacobian products (VJP) with respect to output gradients $\boldsymbol{v}$.

As such, computing the terms $\boldsymbol{v}^T \partial_{\boldsymbol{y}} \beta(\boldsymbol{y};\boldsymbol{\theta})$ and $\boldsymbol{v}^T \partial_{\boldsymbol{\theta}} \beta(\boldsymbol{y};\boldsymbol{\theta})$ can be reduced to computing the component-wise VJPs $v_i \cdot \partial_{y_i} \beta(y_i;\boldsymbol{\theta})_i$ and $v_i \cdot \partial_{\boldsymbol{\theta}} \beta(y_i;\boldsymbol{\theta})_i$. If we have access to the Jacobian diagonal $\operatorname{diag}(\partial_{\boldsymbol{x}} \alpha(\boldsymbol{x};\boldsymbol{\theta}))$, this can be done using eqs. (53) (54) with one VJP call (see next paragraph for an implementation example).

For the log determinant of the jacobian we use

$$\log\det \partial_{\boldsymbol{y}} \beta(\boldsymbol{y},\boldsymbol{\theta}) = \log \prod_i \operatorname{diag}(\partial_{\boldsymbol{y}} \beta(\boldsymbol{y},\boldsymbol{\theta}))_i \tag{81}$$

$$= \sum_i \log \partial_{y_i} \beta(y_i,\boldsymbol{\theta}) \tag{82}$$

$$= \sum_i g_{\beta_i}(y_i,\boldsymbol{\theta}). \tag{83}$$

which reduces computing the VJPs $v \cdot \partial_{\boldsymbol{y}} \log\det \partial_{\boldsymbol{y}} \beta(\boldsymbol{y},\boldsymbol{\theta})$, $v \cdot \partial_{\boldsymbol{\theta}} \log\det \partial_{\boldsymbol{y}} \beta(\boldsymbol{y},\boldsymbol{\theta})$ to computing the component-wise VJPs $v \cdot \partial_{\boldsymbol{y}_i} g_i(y_i,\boldsymbol{\theta})$, $v \cdot \partial_{\boldsymbol{\theta}} g_i(y_i,\boldsymbol{\theta})$. Similarly as before, this can be done with two VJP calls if we have access to the Jacobian diagonal $\operatorname{diag}(\partial_{\boldsymbol{x}} \alpha(\boldsymbol{x};\boldsymbol{\theta}))$ and using eqs. (55), (56) (see next paragraph for an implementation example).

Algorithmic implementation The procedure above is easy to implement in many deep learning frameworks such as `PyTorch`. We give a pseudo-code implementation below. Here `vjp` denotes the vector-jacobian product e.g. as implemented in `PyTorch` via the `torch.autograd.grad` function.

```
def forward_pass(root_finder, bijection, output, params):
    ''' computes forward pass via black-box root finder '''

    # compute inverse via black-box method
    input = root_finder(bijection, output, params)

    # compute forward log det jacobian
    ldj = jacobian_log_determinant(bijection, input, params)

    # return input and corresponding log det jacobian
    return input, -ldj

def backward_pass_x(bijection, output_grad_x, output_grad_ldj, input, params):
```

[2] Here we refer to the terminology introduced in Huang et al. [22]. We do **not** refer to permutation equivariant models leveraging dot-product attention or similar.

```
    ''' computes gradients with respect to inputs '''

    # compute diagonal jacobian and its log
    jac = diagonal_jacobian(bijection, input, params)
    log_jac = log(jac)

    # reweigh output gradients with inverse diagonal jacobian (eqs. 53 & 55)
    output_grad_x = output_grad_x / jac
    output_grad_ldj = output_grad_ldj / jac

    # compute dg/dx
    f = vjp(output=log_jac, input=input, output_grad=ones_like(jac))

    g1 = output_grad_x
    g2 = f * output_grad_ldj

    g = g1 + g2

    return g


def backward_pass_params(bijection, output_grad_x, output_grad_ldj, input, params):
    ''' computes gradients with respect to parameters '''

    # compute output, diagonal jacobian and its log
    output = bijection(input, params)
    jac = diagonal_jacobian(bijection, input, params)
    log_jac = log(jac)

    # reweigh output gradients with inverse diagonal jacobian (eqs. 54 & 56)
    output_grad_x = output_grad_x / jac
    output_grad_ldj = output_grad_ldj / jac

    # compute dg/dx
    f = vjp(output=jac, input=input, output_grad=ones_like(jac))

    # compute sum of eqs. 54 & 56 in one operation
    u = [output, output, jac]
    v = [-output_grad_x, -f * output_grad_ldj, -output_grad_ldj]
    g = vjp(output=u, input=params, output_grad=v)

    return g
```

SI-1.4 Details on multi-bin bisection

Here we give additional details to the multi-bin bisection as sketched in Sec. 5.

Instead of just keeping track of a left and right bound of the search interval (1) we split it into K bins (2) identify the right bin (3) recursively apply the search to this smaller bin. Algorithmically, the procedure is given as follows:

1. Let $[a, b]$ be a closed interval which is known to contain x s.t. $f(x) = y$.
2. For $k = 0 \dots K$ define $s_k = \frac{k}{K}(b - a) + a$.
3. Find k such that $f(s_k) - y < 0$ and $f(s_{k+1}) - y > 0$.
4. If $|f(s_k) - y| < \epsilon$ return $x \approx s_k$. Else set $a := s_k,\ b := s_{k+1}$ and continue with step 2.

We can see that $K = 2$ results in the usual bisection method. However, each step can be executed in vectorized form. Thus for a fixed precision we can reduce the number of iterations by a factor $O\left(\frac{1}{\log K}\right)$ at the expense of increasing memory by a factor $O\left(K\right)$.

SI-2 Additional Details for Experiments

SI-2.1 Illustrative Toy Examples

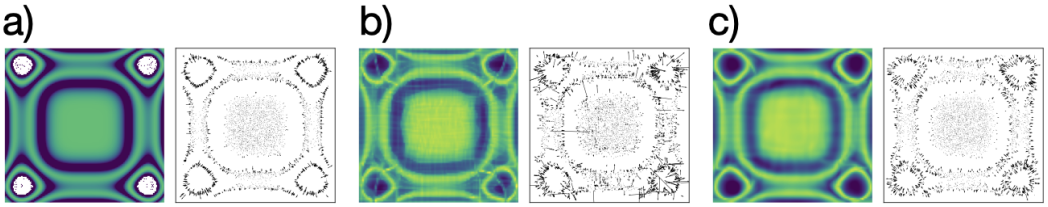

Figure SI-1: a) Reference density and corresponding force field as approximated by b) a C^1-smooth NSF and c) a C^∞-smooth flow using the mixture of bump functions introduced in Sec. 4.

In addition to the simple example presented in Fig. 2 we give another example for the periodic case in Fig. SI-1. As in the non-periodic case, the periodic splines introduce dramatic outliers in the forces, whereas the force field of the C^∞ periodic flows remains smooth.

Compared architecture In both experiments (periodic and non-periodic), we used flows made of four coupling layers where we swapped first and second dimension between each layer. Each conditional component-wise transformation used 40 bins (spline case) or 40 mixture components (smooth cases). For the conditioner networks we used simple dense nets with two hidden layers of size 100 and Swish activations. To satisfy the periodic boundary condition in the periodic case we expand the inputs to these conditioner nets in a cosine basis.

Potentials and data We use two simple potentials allowing us to compute the ground truth density and forces analytically.

For the non-periodic case presented in Fig. 2 we used the energy

$$u(\boldsymbol{x}) = -\log\left(\sum_i \alpha_i \exp\left(-\frac{\|\|\boldsymbol{x}\|_2 - r_i\|_2^2}{2\sigma}\right)\right) \tag{84}$$

with $\sigma = 0.06$ and $\alpha = [1, 0.8, 0.6, 0.4]$.

For the periodic case presented in Fig. SI-1 we used the energy

$$\beta(\boldsymbol{x}) = \left(\cos\left(x_1^2\right) + \cos\left(x_2^2\right) + 2\right)^{\frac{1}{2}} \tag{85}$$

$$u(\boldsymbol{x}) = -\log\left(\sum_i \alpha_i \exp\left(-\frac{\|\beta(\boldsymbol{x}) - r_i\|_2^2}{2\sigma}\right)\right) \tag{86}$$

with $\sigma = 0.05$ and $\alpha = [1, 0.8, 0.6, 0.4]$.

We generated data from u by first running 1,000 Metropolis-Hastings steps using 10,000 parallel chains. The the initial configurations fo the chain were sampled in a regular grid. Then we ran each chain for another 10 steps providing us with 100,000 samples in total.

Training We trained all flows with MLE using a batch size of 1,000 for 8,000 iterations and using Adam [27] optimizer with learning rate of 0.0005.

SI-2.2 Runtime Comparison

Table SI-1 shows the difference in performance between a neural spline flow with analytic inversion and root-finding inversion. For small tensor dimensions, the optimal multi-bin bisection can employ up to 256 bins on the GPU, which results in only a factor of 2-3 slowdown compared to analytic inversion. For larger dimensions, the parallelization over multiple bins becomes less effective, leading to one order of magnitude difference in computational cost.

Table SI-1: Runtimes (in ms) of analytic inversion and root-finding inversion of neural spline flows averaged over 10 runs with 100 evaluations each. K_{opt} denotes the number of bins that yielded the fastest root-finding inversion. The computations were performed on an NVIDIA GeForce GTX1080.

	Inversion				Inversion + Backpropagation			
dim	K_{opt}	analytic	root-finding	factor	K_{opt}	analytic	root-finding	factor
2	156.8 (±54.6)	11.8 (±2.0)	22.6 (±2.2)	2.1 (±0.5)	112.0 (±41.5)	13.3 (±1.7)	25.4 (±2.6)	2.0 (±0.4)
8	60.0 (±30.5)	11.3 (±1.9)	25.0 (±2.1)	2.4 (±0.5)	46.4 (±21.0)	12.4 (±1.5)	24.8 (±2.3)	2.1 (±0.3)
32	27.6 (±9.8)	11.2 (±1.7)	28.3 (±2.7)	2.7 (±0.4)	48.0 (±20.2)	11.8 (±1.9)	31.5 (±2.8)	3.0 (±0.8)
128	14.4 (±4.4)	12.6 (±1.9)	44.1 (±5.4)	3.7 (±0.6)	12.4 (±2.9)	12.1 (±1.5)	40.7 (±4.9)	3.5 (±0.6)
512	6.0 (±1.3)	12.2 (±1.5)	76.0 (±5.3)	6.5 (±1.1)	5.6 (±1.2)	10.6 (±1.9)	75.7 (±3.5	7.8 (±1.5)
2048	4.0 (±0.0)	15.8 (±1.0)	209.8 (±3.6)	13.4 (±0.9)	4.0 (±0.0)	16.1 (±0.9)	211.1 (±4.1)	13.2 (±0.7)

SI-2.3 Alanine Dipeptide Target Energy and Data

The molecular system contained one alanine dipeptide molecule (a 22-atom molecule) in implicit solvent. Its energy function was defined by the Amber ff99SB-ILDN force field with the Amber99 Generalized Born (OBC) solvation model. All covalent bonds were flexible.

Training samples were generated in OpenMM by running molecular dynamics (MD) simulations in the canonical ensemble at 300 K. The 1 μs simulation used Langevin integration with a 1/ps friction coefficient and a 1 fs time step. Coordinates and forces were saved in 1 ps intervals.

The signature feature of the Boltzmann distribution for alanine dipeptide is the joint marginal density over its two backbone dihedrals. This distribution is depicted in the so-called Ramachandran plots in Fig. 3, which are log-scaled heatmaps of the populations. Rotations around these two dihedrals have relatively long transition times and determine the global shape of the molecule, see the exemplary samples in Fig. 3 a).

SI-2.4 Alanine Dipeptide Boltzmann Generator

The normalizing flow was set up in a physically informed way, see Fig. SI-2. All learnable transforms operated in an internal coordinate framework, where deeper parts of the network corresponded to torsion angles. This is a natural choice, as rotation around torsion angles determine the global structure of the molecules, while bond lengths and 1-3 angles are comparatively stiff.

The base density for intramolecular bond lengths, angles, and torsion angles is defined as uniform distributions on the unit hypercube, $\mathcal{U}\left([0,1]^{21} \times [0,1]^{20} \times [0,1]^{19}\right)$. The latent variables corresponding to torsion angles were split into two input channels. In the first 8 coupling layers, the two torsion channels were conditioned on each other using either neural spline transforms or mixtures of smooth bump functions. After this transformation, the two channels were merged back into a single torsion channel. Bond and angle channels were conditioned on each other using four subsequent coupling layers. Finally, the angle channel was conditioned on all torsions and the bond channel on all angles and torsions in an autoregressive manner.

The number of bins and components was 8 for spline and bump transforms, respectively. All elementwise transforms were defined as bijections on the unit interval, where torsions were considered as circular elements. All conditioners had two fully connected hidden layers with 64 features each and sin activations. All circular degrees of freedom (torsions) were mapped onto the unit sphere before being passed into the dense conditioner network. Torsional degrees of freedom were transformed with circular transforms, bonds and angles with non-circular transforms.

The final two layers were designed as follows. First, all IC channels were mapped from $[0, 1]$ to their respective domains by fixed linear transformations, where bonds and angles were constrained

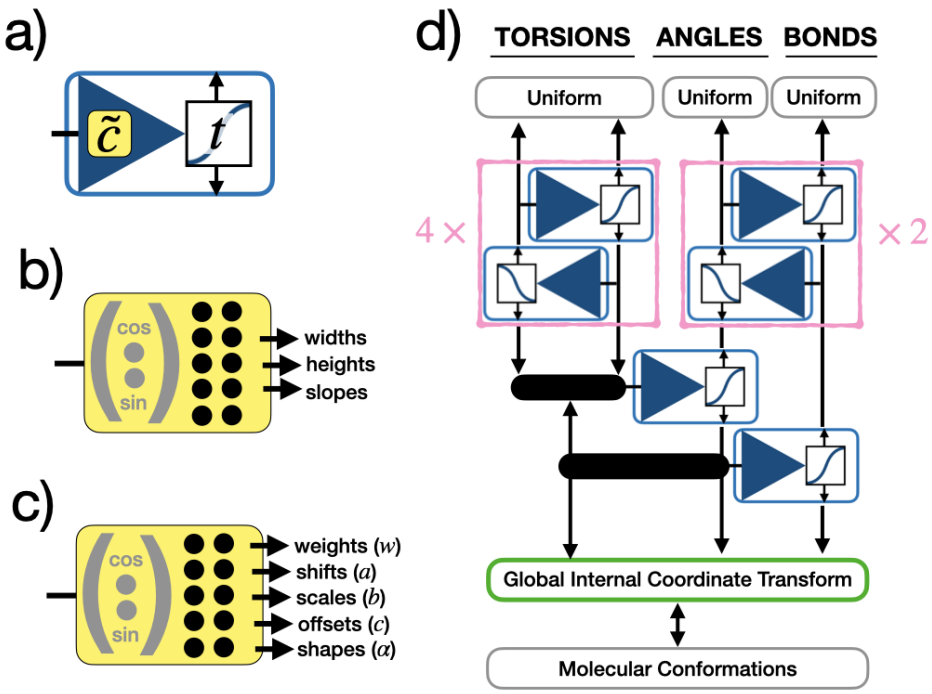

Figure SI-2: Network architecture – (a) Coupling block: a dense conditioner network $\tilde{c}$ generates shape parameters that define an elementwise bijective transform t from the first input channel. The transform is then applied to the second input channel. (b) Conditioner network for spline transforms returning bin widths, bin heights, and slopes at the support points. Circular inputs are first wrapped around the unit circle. For circular outputs, the first and last slope are enforced to be identical. (c) Conditioner network for smooth transforms (mixtures of bump functions) returning the shape parameters for each bump function. (d) Boltzmann generator architecture for alanine dipeptide using coupling blocks as in (a). The global translation and rotation required for the coordinate transform are fixed. Torsions are circular, angles and bonds are not.

to physically reasonable values, [0.05 nm, 0.3 nm] for bonds and $[0.15\pi, \pi]$ for angles, which bracket the values encountered in the data. Finally, the ICs were transformed to Cartesian coordinates through a global internal coordinate transform. The global origin and rotation of the system were fixed in the forward pass and ignored in the inverse pass. The flows with spline transformers had 398,691 parameters in total. The flows with smooth transformers had 758,168 parameters as the shape parameters of the bump functions are also learnable.

SI-2.5 Boltzmann Generator Training

The loss function (5) was minimized using 90% of the data set and a batch size of 128. The inversion of smooth flows was performed with multi-bin root finding and $K = 100$ bins. The Adam optimizer [27] was initialized with a learning rate 5×10^{-4} and the learning rate was scaled by 0.7 after each epoch. Optimization was run for 10 epochs. For validation, the negative log likelihood, force matching error, and reverse KL divergence (up to an additive constant) were computed over the remaining 10% of the dataset.

Singularities of the target energy function can negatively affect energy-based training. To stabilize the computation of the reverse KL divergence, the loss was cut off for exploding energies. Concretely, the reverse KL divergence was replaced by $\Lambda\left(\mathcal{L}_{\mathrm{KLD}}(\boldsymbol{\theta})\right)$ with

$$\Lambda(x) := \min\left\{x,\ 10^3 + \log(1 + x - 10^3),\ 10^9\right\}.$$

The training runs for the comparison in Table 1 were done over a reduced dataset with only every tenth trajectory snapshot present in both the train and test set and with a constant learning rate of 5×10^{-4} over 10 epochs.

SI-2.6 Affine Boltzmann Generator

To compare with a previously published method, a Boltzmann generator with affine transforms was trained using density estimation and the same hyperparameters as the smooth flow depicted in Fig. 3. In comparison, the affine flow does not as cleanly separate modes on the torsion marginal and produces inaccurate forces and energies.

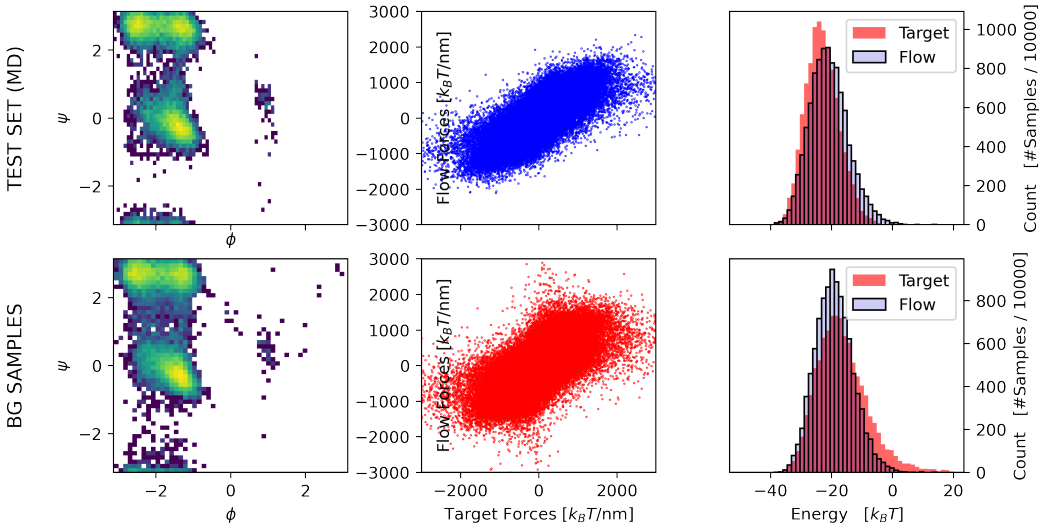

Figure SI-3: Boltzmann generators with affine (RealNVP) transforms trained through likelihood maximization. See Fig. 3 (c) for comparison.

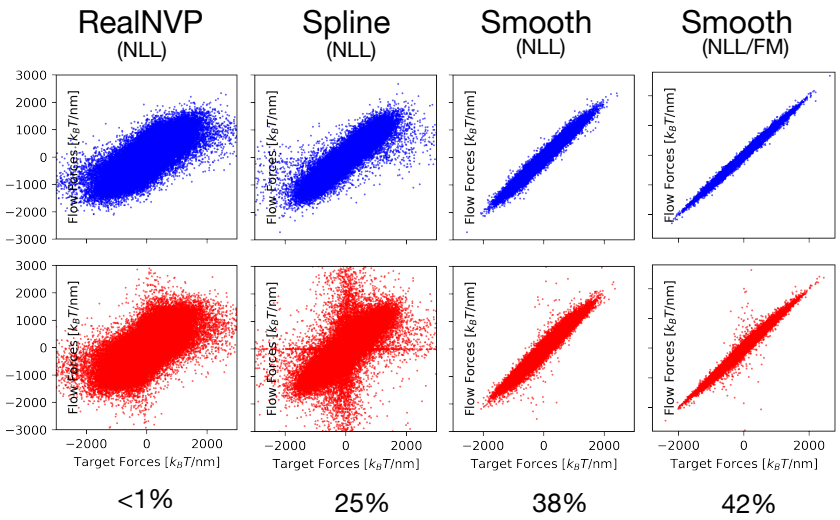

Figure SI-4: Force residues on the test set (upper row) and on BG samples (lower row) for different flow transforms and training methods. The numbers in the last row denote the sampling efficiency.

Figure SI-4 compares forces and sampling efficiency between Boltzmann generators using different types of transforms and training methods via the protocol described in Section SI-2.5. The sampling efficiency is evaluated as the Kish effective sample size divided by the number of samples.

SI-2.7 Molecular Simulations

MD simulations of alanine dipeptide were run for three different potential energy functions. The first potential was the molecular target energy as defined by the Amber ff99SB-ILDN force field with the OBC implicit solvent model. As typical for MD potentials, the energy is defined as a sum over individual terms that correspond to covalent bond stretching, angle bending, rotation around proper and improper dihedrals, pairwise Coulomb and Lennard-Jones interactions, as well as an Generalized-Born implicit solvent term that takes into account the solvent-accessible surface area of each atom and the solvent dielectric constant (78.5 for water). The energy and forces were evaluated using OpenMM.

The other two potential energies were defined by normalizing flows that were trained on the full dataset for 10 epochs each. The energy of a flow is given by $u_\theta = -\log p_f(\cdot; \boldsymbol{\theta})$, where p_f denotes the push-forward distribution of the prior under the learned bijection f. Both flows used the same architecture except all flow transforms were smooth transforms (mixtures of 8 smooth bump functions) in one case and neural spline transforms (rational-quadratic splines [42, 54] with 8 bins) in the other case. The smooth flow was the one described in Sec. 6.3, which was trained by a combination of force matching and MLE. The spline flow was trained by MLE only since spline flows trained with a mixed loss performed poorly on the validation set (cf. Table 1). Otherwise, all hyperparameters were identical. From each training, the flow with the smallest validation loss was selected for the MD simulation.

The first 10 samples from the dataset were chosen as starting configurations. Initial velocities were sampled randomly from a Maxwell-Boltzmann distribution at 300 K. First, 1000 integration steps were performed with a Langevin integrator in each potential to equilibrate the kinetic energy. Next, the thermostat was removed and the system was propagated for 5000 steps using a velocity Verlet integrator, which keeps the total energy constant up to numerical errors.

Consequently, the energy fluctuation and drift are a measure for numerical errors accumulated in the simulations. For the classical force field, these errors arise primarily from the fastest degrees of freedom (in this case the covalent hydrogen bond vibration) that are not sufficiently resolved by the 1 fs time step. The fact that smooth flows exhibit energy fluctuations in a similar range as the MD force field indicates that the roughness of the potential energy surface is comparable.

SI-3 Computing Requirements

The MD simulation and network training was performed on an in-house cluster equipped with NVIDIA GeForce GTX 1080 GPUs. The net runtime for all sampling and training reported in this work was approximately (30 experiments x 10 replicas x 5h) = 1500 h.

SI-4 Used Third-Party Software

All our models were implemented in `PyTorch` [39]. For the Neural Spline Flows, we used the `nflows` library provided by Durkan et al. [13]. We further used `OpenMM` [14] for the simulations and `mdtraj` [33] for analyzing the MD data. Beyond this we used `NumPy`[20] for all non-GPU related numerical computations and `Matplolib` [23] for plotting.