Constrained Transformations in Stan and Jax

Michael Issa

February 2025

Table of contents

1 General Setup

```python
import matplotlib.pyplot as plt
import logging
import cmdstanpy
from cmdstanpy import CmdStanModel
import pandas as pd
import numpy as np
import warnings
import arviz as az
import os

warnings.filterwarnings("ignore")

# Graphic configuration
c_light = "#DCBCBC"
```

```python
c_light_highlight = "#C79999"
c_mid = "#B97C7C"
c_mid_highlight = "#A25050"
c_dark = "#8F2727"
c_dark_highlight = "#7C0000"

c_light_teal = "#6B8E8E"
c_mid_teal = "#487575"
c_dark_teal = "#1D4F4F"

RANDOM_SEED = 58583389
np.random.seed(RANDOM_SEED)
az.style.use("arviz-whitegrid")

plt.rcParams['font.family'] = 'serif'

plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.titlesize'] = 12

plt.rcParams['axes.spines.top'] = False
plt.rcParams['axes.spines.right'] = False
plt.rcParams['axes.spines.left'] = True
plt.rcParams['axes.spines.bottom'] = True

plt.rcParams['axes.xmargin'] = 0
plt.rcParams['axes.ymargin'] = 0

plt.subplots_adjust(left=0.15, bottom=0.15, right=0.9, top=0.85)

current_working_directory = os.getcwd()

cmdstanpy_logger = logging.getLogger("cmdstanpy")
cmdstanpy_logger.disabled = True
cmdstanpy.install_cmdstan(compiler=True)
```

```
CmdStan install directory: C:\Users\issam_biodcm6\.cmdstan
CmdStan version 2.36.0 already installed
Test model compilation
```

```
True
```

```
<Figure size 720x480 with 0 Axes>
```

This is part 1 of a series of documents charting the progression of my work on JaxGPStuff. I've always found it helpful to take the time to write out the actual content of what a specific part of a program is doing in a long-form cogent manner (something I think isn't done enough if at all), and it's something I immensely appreciate (I don't enjoy digging through your raw codebase to find out what you've done).

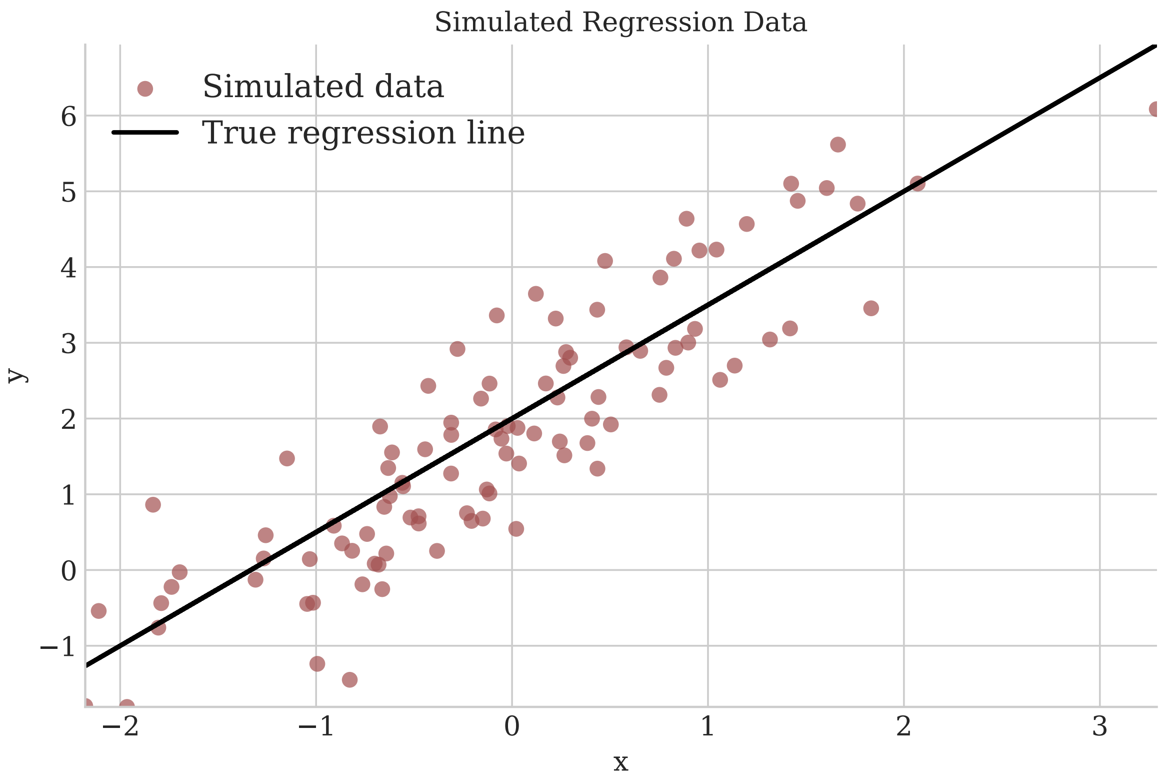
2 Constrained Transformations in Stan

We'll start out easy with an example of in Stan and explain what it is Stan is doing by constraining parameters.

First we'll simulate some simple regression data.

```python
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Simulate data
n = 100
alpha = 2
beta = 1.5
sigma = 0.8

x = np.random.normal(loc=0, scale=1, size=n)

error = np.random.normal(loc=0, scale=sigma, size=n)
y = alpha + beta * x + error

sim_data = pd.DataFrame({'x': x, 'y': y})

x_reshaped = x.reshape(-1, 1)
model = LinearRegression()
model.fit(x_reshaped, y)

print("Intercept (alpha):", model.intercept_)
print("Coefficient (beta):", model.coef_[0])
print("Mean squared error:", mean_squared_error(y, model.predict(x_reshaped)))

plt.scatter(x, y, label="Simulated data", alpha=0.7, color=c_mid_highlight)
plt.plot(x, alpha + beta * x, color="black", label="True regression line", linewidth=2)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Simulated Regression Data")
plt.legend(loc="upper left")
plt.show()
```

```
Intercept (alpha): 1.8796387702452613
Coefficient (beta): 1.5259117159172857
Mean squared error: 0.6303553871943935
```



In statistics and maths, we have various constraints on some parameters values. Variances or standard deviations must be positive, probabilities must be between 0 and 1, correlation matrices must have eigenvalues in valid ranges, etc. Some of them we know are constrained to be positive because they are so defined like the standard deviation. Other times, we know, if these parameters are being intepreted or are structured in some way, that these parameters must be positive. Assume we don't already know what our regression parameters values are, as we usually don't. But we have some idea that they are positive because maybe the intercept stands for average height and there are no negative height values so no negative mean heights. This is a constraint imposed by the problem we're studying.

In Stan, we specify constraints in various ways (see the variables declaration section for the syntax specification) but typically you'll see <"lower=0"> following the function argument type. This expresses a constraint of positivity on whatever parameter it attaches to. Here's a Stan model used to fit our simulated data that uses it for all parameter delcarations.

Stan
Program 1 `model1.stan`

```
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
}

parameters {
    real<lower=0> alpha;
    real<lower=0> beta;
    real<lower=0> sigma;
}

model {
    y ~ normal(alpha + beta * x, sigma);
}
```

Now, it's quite opaque what exactly the constraints are doing here when the computation occurs, but we can explictly represent the constraints Stan computes using the simple transformations Stan is using under the hood. Here is the model without the constraint syntax but constrained by explicit syntactic representation of the computation and by updating the log probability manually.

We fit both our models and look at the output summary now.

```
data = {
    'N': n,
    'x': x.tolist(),
    'y': y.tolist(),
}
```

```
model1 = CmdStanModel(stan_file='models/model1.stan')
fit1 = model1.sample(data=data, chains=4, iter_sampling=1000, iter_warmup=500, show_progress

fit1.summary()
```

Stan
Program 2 `model1_explicit.stan`

```
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
}

parameters {
    real alpha_unc;
    real beta_unc;
    real sigma_unc;
}

transformed parameters {
    real alpha = exp(alpha_unc);
    real beta = exp(beta_unc);
    real sigma = exp(sigma_unc);
}

model {
    target += alpha_unc;
    target += beta_unc;
    target += sigma_unc;

    y ~ normal(alpha + beta * x, sigma);
}
```

	Mean	MCSE	StdDev	MAD	5%	50%	95%	ESS_bulk	ESS
lp__	-27.622300	0.027799	1.214040	1.002390	-30.073900	-27.317500	-26.280500	1976.00	2405
alpha	1.877840	0.001274	0.081531	0.080550	1.743550	1.877650	2.012280	4108.88	2843
beta	1.525980	0.001304	0.081460	0.082344	1.392380	1.524590	1.661810	3924.87	2613
sigma	0.809912	0.001003	0.059192	0.056371	0.718451	0.806936	0.912513	3579.25	2652

```python
model1_explicit = CmdStanModel(stan_file='models//model1_explicit.stan')
fit2 = model1_explicit.sample(data=data, chains=4, iter_sampling=1000, iter_warmup=500, show_

fit2.summary()
```

	Mean	MCSE	StdDev	MAD	5%	50%	95%	ESS_bulk
lp__	-27.566200	0.027273	1.182140	0.968434	-29.892600	-27.255300	-26.265100	2050.02
alpha_unc	0.629589	0.000655	0.041943	0.041334	0.559548	0.631160	0.697422	4156.59
beta_unc	0.420922	0.000804	0.052845	0.053089	0.331678	0.423594	0.505266	4369.39
sigma_unc	-0.210302	0.001363	0.071842	0.070495	-0.323411	-0.212575	-0.088289	2813.00
alpha	1.878490	0.001227	0.078695	0.077755	1.749880	1.879790	2.008570	4156.60
beta	1.525490	0.001216	0.080304	0.081024	1.393310	1.527440	1.657430	4369.41
sigma	0.812439	0.001120	0.058713	0.057120	0.723676	0.808500	0.915497	2813.03

We see the two Stan models provided are mathematically equivalent and will recover the same parameter values because they involve two closely identical computations of the same reparameterization, a constrain of positivity. The key difference between them lies in where the code is being represented. Stan does it somewhere else as a subroutine not represented to the user, but we did it directly. We'll break the math down in a bit more detail now.

In the first model, the parameters α, β, σ are constrained to be greater than or equal to 0 (i.e., α, β, and $\sigma \geq 0$). This means the model explicitly ensures these parameters are non-negative during sampling.

Stan will sample these parameters directly in their constrained space (keep in mind: sampling in the constrained space is NOT equivalent to sampling and transforming your sample to be constrained. We'll revisit this below). The parameter values are bounded between 0 and ∞. This means the likelihood will incorporate the probability of the parameters being within the defined range, i.e., using the normal distribution for the regression model, along with the boundary conditions on the parameters (such as $\sigma \geq 0$).

Both models are mathematically[1] equivalent because the transformations in the second model enforce the same constraints on the parameters as the first model but instead of performing the adjustment on the log density automatically as Stan does if you specify bounds, we do it by hand (Why increment the target probability by the value X_{unc} you might ask? Keep reading). The transformation $\exp(x)$ maps all real values of α_{unc}, β_{unc}, and σ_{unc} to positive values, which is what the first model does by directly imposing constraints $\alpha, \beta, \sigma \geq 0$.

[1]Not computationally equivalent. Besides that point, it's also not true that a change of variables preserves the log density. It would defeat the purpose if it did, since the transformation defines a probability density, which is a non-unique, parameterization-dependent function in $\mathbb{R}^N \to \mathbb{R}_+$. In practice, this means a given model can be represented different ways, and different representations have different computational performances. There are at least two levels of representation of a model going on here. One is at the low level (where we are) where we have changes in the log density due to various computations performed on it. The other level is one we'll get to in the next section where we reparameterize a model by composing it with some functions to transform the sampled outputs. This is what you typically do, for example, in reparameterizing a beta distribution with the mean and total counts rather than in terms of the two counts, α and β. This reparameterization doesn't distort the underlying log density when you sample because it's done after the fact.

- In the first model, the parameter space is constrained to be non-negative directly, so Stan only samples within the range $[0, \infty]$.
- In the second model, the parameters are unconstrained, and the exponential transformation ensures that the resulting parameters are positive. The log of the Jacobian $\log\left(\left|\frac{d}{dx}\exp(x)\right|\right)$ is then added to the target log-probability to adjust for the transformation from unconstrained space to constrained space because we want to interpret our sampling in the original space.

Both methods perform close to identical operations[2] for the log densities and will therefore produce the same parameter values after sampling. This is because both models are doing the same reparameterization.

More precisely, the likelihood function in both models is:

$$y_i \sim \mathcal{N}(\alpha + \beta x_i, \sigma)$$

In the second model, the unconstrained parameters $\alpha_{\text{unc}}, \beta_{\text{unc}}, \sigma_{\text{unc}}$ are mapped to positive values via the exponential function:

$$\alpha = \exp(\alpha_{\text{unc}}), \quad \beta = \exp(\beta_{\text{unc}}), \quad \sigma = \exp(\sigma_{\text{unc}})$$

The Jacobian of this transformation is:

$$\frac{d}{d\theta}\exp(\theta) = \exp(\theta)$$

So, the log-Jacobian adjustment is:

$$\log(\exp(\alpha_{\text{unc}})) = \alpha_{\text{unc}}, \quad \log(\exp(\beta_{\text{unc}})) = \beta_{\text{unc}}, \quad \log(\exp(\sigma_{\text{unc}})) = \sigma_{\text{unc}}$$

Thus, the log-probability is adjusted by adding these terms to account for the transformation from the unconstrained space to the constrained space. Stan typically does these change of variable transformations automatically using a more generalized transformation for the lower-bound constraint. But I find it quite nice that Stan allows you to program things manually and increment the log density using target+=. You might ask why would I ever care to figure out what the log Jacobian adjustment is when Stan can do it for me?[3] You don't ever need to if all you're doing is specifying constraints like the ones Stan provides for variables, but there are constraints that you would want to impose that do require it. There are too many reasons you'd want to that I'm unaware of but here are a few: high posterior correlation in unconstrained space (cf. Neal's funnel), numerical stability, prior stability, and sampling

[2]Not exactly. Stan uses the general transformation $X = \exp(Y) + a$ for a lower bound constraint being a.

[3]Besides the fact that it's nice to know you can do it and you should know what Stan is doing!

efficiency. It would take too long to go into the details and Google is your best friend here. Outside of Bayesian inference related things, constrained transformation have found a home in some kick ass generative models that are based on Normalizing Flows.[4]

Looking back at our models you might wonder if these non-linear transformation applied are actually changing the underlying density substanitally. You'd be right to think this. For our example, we can refit the model without the Jacobian adjustment and see what happens.

Stan
Program 3 `model1_explicit.stan`

```
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
}

parameters {
    real alpha_unc;
    real beta_unc;
    real sigma_unc;
}

transformed parameters {
    real alpha = exp(alpha_unc);
    real beta = exp(beta_unc);
    real sigma = exp(sigma_unc);
}

model {
    y ~ normal(alpha + beta * x, sigma);
}
```

```
model1_explicit_no_increment = CmdStanModel(stan_file='models//model1_explicit_no_increment.s
fit3 = model1_explicit_no_increment.sample(data=data, chains=4, iter_sampling=1000, iter_warn

fit3.summary()
```

[4]Again Google.

	Mean	MCSE	StdDev	MAD	5%	50%	95%	ESS_bulk
lp___	-28.465300	0.027567	1.255610	1.004020	-30.902100	-28.137100	-27.114300	2184.55
alpha_unc	0.628232	0.000724	0.044194	0.044910	0.555909	0.628785	0.699591	3742.07
beta_unc	0.417134	0.000811	0.054371	0.054282	0.326635	0.417550	0.504227	4536.13
sigma_unc	-0.213706	0.001194	0.070709	0.070322	-0.326706	-0.215641	-0.094641	3567.97
alpha	1.876120	0.001359	0.082917	0.084278	1.743530	1.875330	2.012930	3742.04
beta	1.519840	0.001227	0.082412	0.082573	1.386290	1.518240	1.655700	4536.21
sigma	0.809614	0.000980	0.057637	0.056042	0.721296	0.806024	0.909699	3567.97

Nothing seems to have changed except a decrease in the log probability. The underlying density doesn't exactly match anymore. But, this doesn't mean our inference goes totally awary. The parameters estimates match our first two model estimates because the underlying geometry wasn't distorted by much. We shouldn't expect it to for a simple model where the data is fairly well behaved. We're also not applying the transformation on our observed output value y to expect much of a change, and the likelihood term isn't affected as much by the transformations of the parameters because we have enough data for the estimation to find the posterior mean easily. Even if we had substantially less data in our case we would still be able to find it because we're conducting a simple transformation on parameters we know are positive.

3 A Note on Change of Variables VS. Transformations

Smarter people than I have tried to drill this distinction down. I highly, highly recommend reading and rereading the Stan documentation here. They give many examples at varying complexity and really hammer home the difference.

To put it succietnly, change of variables transformation are transformations that first transform the variables AND THEN sample them. Whereas transformations are typically taken to be instances where we sample some variables AND THEN apply a functional transformation to them. Again, read that Stan article. It provides many examples that demonstrate the difference.

Before moving onto the architecture for our own transform class, I'll plug in two nice articles. One of them by Jacob Socolar clarifies the nature of the Jacobian adjustment with a simple example. He links to various soruces on the same topic in there that are well worth a look. The other post is by Aki Vehtari and gives a nice treatment of the Laplace approximation and Jacobian adjustment, going slighly more in depth. I said two but I can't leave out this article by Michael Betancourt that does go through the mathematical details in the section linked. Socolar links to it and *highly recommends* it. As do I. Betancourt's work is always top-tier.[5]

[5]LOVE his work. Best stuff out there! Read it and buy him a coffee here if you feel you got something out of it.

4 Some Quick Mathematics for Probability and Constrained Transformations

"It would be good to cover the foundations of continous probability theory. Start with a space consisting of an ambient set X, endowed with a σ-algebra $\mathcal{X}$ consisting of all well-behaved subsets of X and a probability distribution π that maps elemens of the σ-algebra into probabilities in a way that's comptabile with countable unions, intersections, and complements."[6]

When working with probability theory, particulary on $\mathbb{R}^n$, the choice of σ-algebra is crucial. The two most common σ-algebras are (1) Borel σ-algebra, $\mathcal{B}(\mathbb{R}^n)$, generated by all open sets in $\mathbb{R}^n$. These include intervals, rectangles, and their countable operations. And (2) Lebesgue σ-algebra, $\mathcal{L}(\mathbb{R}^n)$, which are an extension of the Borel σ-algebra that includes more sets. The relationship between these is:

$$\mathcal{B}(\mathbb{R}^n) \subset \mathcal{L}(\mathbb{R}^n)$$

$\mathcal{L}(\mathbb{R}^n)$ contains all Borel sets plus all subsets of Borel sets with Lebesgue measure zero. When considering transformations between spaces, these distinctions become important: measurable functions must preserve the σ-algebra structure.

"When considering another space Y equipped with its own σ-algebra $\mathcal{Y}$ along with a map $F : X \to Y$, this point-wise mapping can induce maps between objects defined on these spaces. The original map F induces both a pushforward map (in the same direction of F) and a pullback map (in the opposite direction of F) between subsets on X and Y. If the pullback map is compatible with the σ-algebras so that for every $B \in \mathcal{Y}$ we have $F^{-1}(B) \subset \mathcal{X}$, then we can define an induced pushforward map between probability distributions. Every probability distribution π defined on X defines a pushforward probability distribution F_π on Y via $P_{F_\pi}[B] = P_\pi[F^{-1}(B)]$.

Now let's consider another space Y equipped with its own σ-algebra $\mathcal{Y}$ along with a map $F : X \to Y$.

Nominally F just maps points in X to points in Y but this point-wise mapping can also induce maps from objects defined on X to objects defined on Y. For example by breaking a subset $A \subset X$ into points and then mapping them to Y before collecting those output points in other subset $F(A) \subset Y$ the original map F induces a map from subsets on X to subsets on Y. This kind of induced map in the same direction of F is called a *pushforward* along F.

At the same time F might also induce maps from objects defined on Y to objects defined on X. If F isn't bijective then we can't define an inverse point-wise map $F^{-1} : Y \to X$, but we can we can define a map from subsets $B \subset Y$ to subsets $F^{-1}(B) \subset X$. This kind of induced map in the opposite direction of F is called a *pullback* along F.

[6] Betancourt

So the point-wise map F induces both a pushforward and pullback map between subsets on X and Y. These induced maps, however, will not in general respect the σ-algebras. In particular if $A \in \mathcal{X}$ then the output of the pushforward map $F(A)$ need not be in $\mathcal{Y}$, and vice versa for the pullback map.

If the pullback map is compatible with the σ-algebras so that for every $B \in \mathcal{Y}$ we have $F^{-1}(B) \subset \mathcal{X}$ then we can define *another* induced pushforward map, this time between probability distributions. Every probability distribution π defined on X defines a pushforward probability distribution $F_* \pi$ on Y via the probabilities

$$\mathbb{P}_{F_* \pi}[B] = \mathbb{P}_{\pi}[F^{-1}(B)].$$

Again we need $F^{-1}(B)$ to be in $\mathcal{X}$ otherwise the initial probability distribution won't know how to assign a probability to the pullback subset.

Measurable functions/maps/transformations are just the maps satisfying the compatibility requirement that allows us to define pushforward probability distributions. In other words measurable maps are the only maps that allow us to translate probability distributions from one space to another.

Note that at this point no other requirement has been made on the structure of X, Y, and F. X and Y don't have to have the same dimensions, F doesn't have to be bijective or even injective so long as it satisfies the σ-algebra consistency property.

If the dimension of Y is less than the dimension of X then a measurable surjection $F : X \to Y$ is commonly known as projection map, and pushforward distributions are known as *marginal* distributions.

If the dimension of X and Y are the same and both F and F^{-1} are measurable then a bijection $F : X \to Y$ is commonly known as a reparameterization.

(Side note: codomains are irrelevant here as the σ-algebras and probability distributions of interest are all defined over the entire domain).

They key difference between these two types of maps is that projections loose information while reparameterizations do not. If F is a reparameterization then we can start at π on X, pushforward to $F_* \pi$ on Y, then pushforward along F^{-1} to recover the original distribution,

$$(F^{-1})_* F_* \pi = \pi.$$

This is not true of projection functions – we can map π on X to $F_* \pi$ on Y but there's no way to recover π from that pushforward distribution.

Okay, so now we're *finally* ready to talk about probability density functions. Probability density functions are functions that quantify the difference between two measures. Mathematically we denote the density function of π_2 with respect to π_1 as

$$\pi_{21}(x) = \frac{\mathrm{d}\pi_2}{\mathrm{d}\pi_1}(x).$$

Most often we correct some standard "uniform" distribution on the ambient space to the probability distribution of interest. If X is a real space then that uniform distribution is the Lebesgue measure, $\mathcal{L}$. In other words the probability density function of π is actually the probability density function of π relative to the Lebesgue measure,

$$\pi(x) = \frac{\mathrm{d}\pi}{\mathrm{d}\mathcal{L}}(x).$$

Using the above machinery we can in some cases work out how to construct pushforward probability density functions. The basic idea is to take a distribution on X, push it forward along F to $F_*\pi$ on Y and then construct the density of each with respect to the uniform measures on X and Y respectively. In other words

$$\pi(x) = \frac{\mathrm{d}\pi}{\mathrm{d}\mathcal{L}_X}(x) \mapsto \pi(y) = \frac{\mathrm{d}F_*\pi}{\mathrm{d}\mathcal{L}_Y}(y).$$

Notice that we pushforward π along F but we define the densities with respect to the uniform distributions on X and Y respectively. We don't transform the uniform distribution on X to some distribution on Y because that pushforward distribution will in general no longer be uniform! Indeed when $F : X \to Y$ is a measurable bijection the amount by which F warps the initial uniform distribution is just the Jacobian determinant!

Mathematically when F is a bijection we can write

$$\pi(y) = \frac{\mathrm{d}F_*\pi}{\mathrm{d}\mathcal{L}_Y}(y) = \frac{\mathrm{d}F_*\pi}{\mathrm{d}F_*\mathcal{L}_X}(y) \cdot \frac{\mathrm{d}F_*\mathcal{L}_X}{\mathrm{d}\mathcal{L}_Y}(y) = \pi(F^{-1}(y)) \cdot |J|(y)$$

which is exactly the usual "change of variables" formula that's pulled out of thin air.

When F is a surjection then the density of the pushforward uniform distribution from X relative to the uniform distribution on Y, $\mathrm{d}\mathcal{L}_X/\mathrm{d}\mathcal{L}_Y$ is singular and so the usual change of variables formula cannot be applied. In these cases working out the pushforward probability density functions, or the marginal density functions, is much, much harder and usually cannot be done analytically."[7]

[7] Betancourt

5 Contraint Transformations in GPJaxStuff

Why should we care about contraint transformations for Gaussian Processes? We can enforce various kinds of contraints on our mean function, kernel outputs, kernel parameters, or predictions. GPs are glorified priors on function spaces. We should be constraining that space as much as possible. The structure of our