Constrained Transformations in Stan and Jax

Michael Issa

February 2025

Table of contents

1 General Setup

```
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"
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

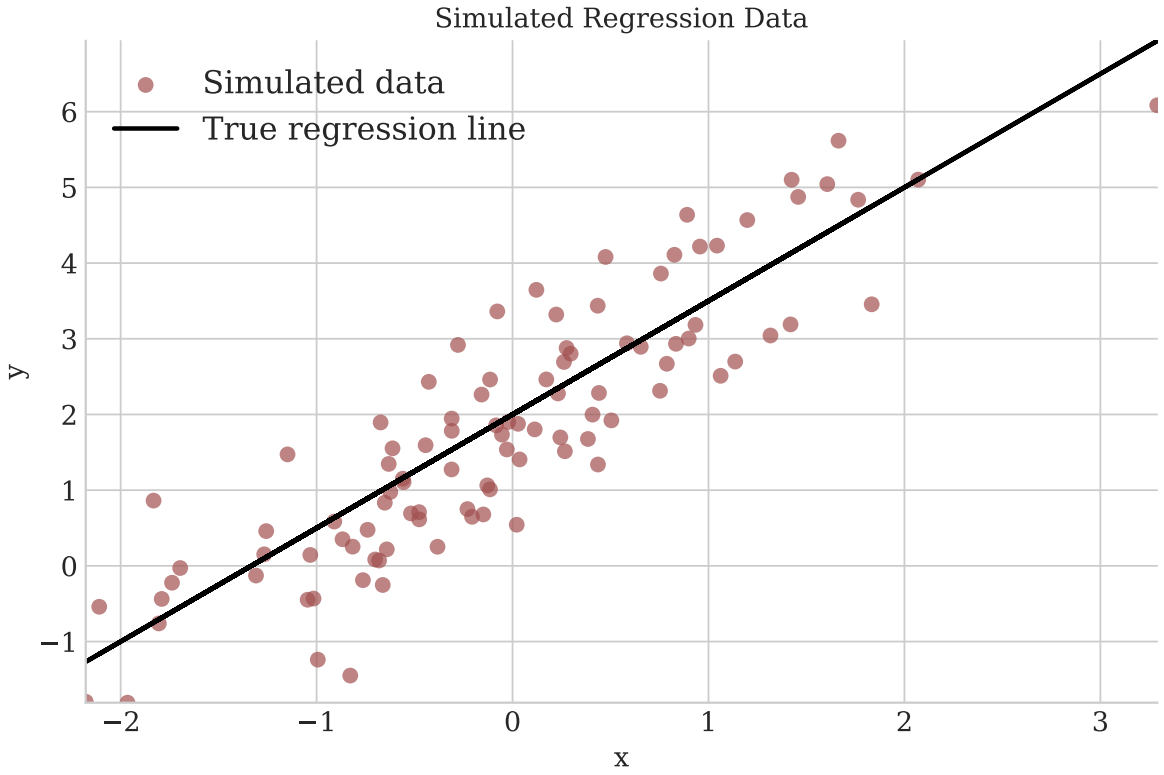
We'll start out easy with an example of in Stan and explain what it is Stan is doing by constraining parameters.

First we'll simulate some simple regression data. 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.

```
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 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___	-27.548300	0.026476	1.198950	0.964209	-29.837700	-27.240700	-26.259800	2147.79	2558
alpha	1.879680	0.001388	0.079960	0.078289	1.746770	1.878510	2.011470	3346.63	2709
beta	1.525990	0.001338	0.079584	0.076391	1.393150	1.525290	1.657350	3544.73	2811
sigma	0.813226	0.000993	0.057706	0.055225	0.724179	0.809241	0.914189	3491.32	2747

```
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___	-27.591200	0.027615	1.224010	0.990822	-30.099600	-27.270100	-26.264700	1983.21
alpha_unc	0.629911	0.000751	0.042230	0.041859	0.558216	0.631891	0.696182	3311.64
beta_unc	0.420838	0.000920	0.054107	0.053601	0.329447	0.422932	0.508122	3538.02
sigma_unc	-0.209985	0.001252	0.071872	0.071369	-0.326586	-0.210925	-0.089854	3311.98
alpha	1.879110	0.001392	0.079052	0.078400	1.747550	1.881160	2.006080	3311.64
beta	1.525460	0.001388	0.082199	0.081291	1.390200	1.526430	1.662170	3538.04
sigma	0.812696	0.001027	0.058660	0.057593	0.721382	0.809835	0.914065	3312.01

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 $\alpha, \beta, \sigma \geq 0$.

- 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 efficiency. It would take too long to go into the details and Google is your best friend. Outside of Bayesian inference related things, constrained transformation have found a home in some kick ass generative models that are based on Normalizing Flows.[1]

3 A note on change of variables transformation vs transformations

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

[1] Again Google.

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.

[^1] NOT COMPUTATIONALLY EQUIVALENT.

[^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!

It would defeat the purpose if they were, and 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.