MCMC - Performance & Stan

Lecture 20

Dr. Colin Rundel

`.pytensorrc` setup

On the departmental server, to avoid missing BLAS library warnings with PyMC / PyTensor, create `~/.pytensorrc` with the following contents:

```
[blas]
ldflags = -lflexiblas
```

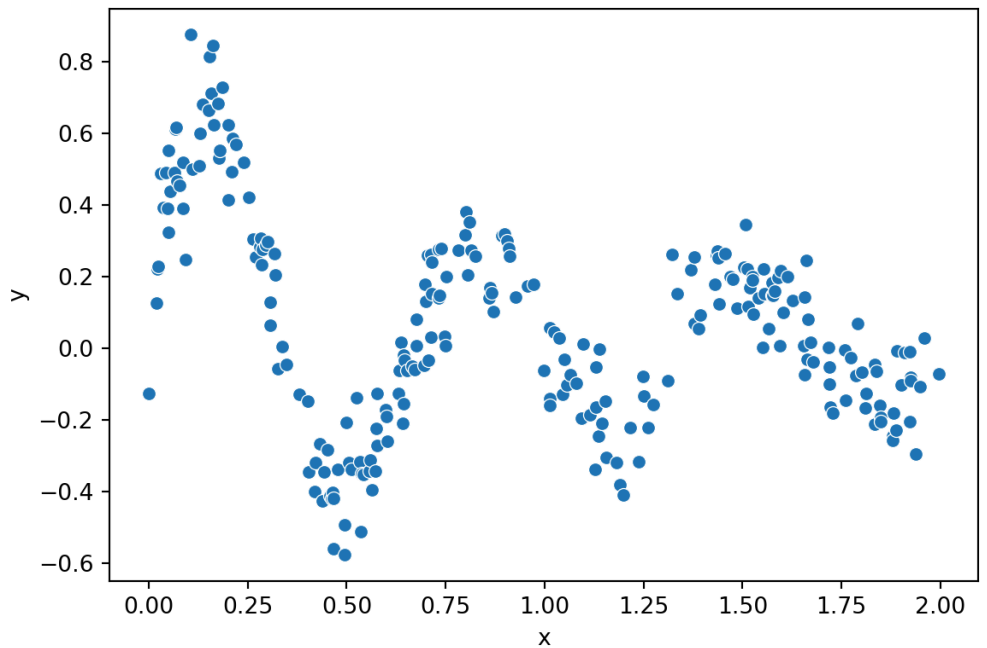
Example - Gaussian Process

Data

```
1 d
```

```
            x         y
0    0.000258 -0.126679
1    0.019467  0.125848
2    0.021670  0.222179
3    0.022712  0.228689
4    0.029758  0.487060
..        ...       ...
245  1.925221 -0.090028
246  1.936691 -0.295139
247  1.948376 -0.106351
248  1.958606  0.029466
249  1.995473 -0.071615

[250 rows x 2 columns]
```



GP model

```
X = d.x.to_numpy().reshape(-1,1)
y = d.y.to_numpy()

with pm.Model() as model:
  l = pm.Gamma("l", alpha=2, beta=1)
  s = pm.HalfCauchy("s", beta=5)
  nug = pm.HalfCauchy("nug", beta=5)

  cov = s**2 * pm.gp.cov.ExpQuad(input_dim=1, ls=l)
  gp = pm.gp.Marginal(cov_func=cov)

  like = gp.marginal_likelihood(
    "y", X=X, y=y, sigma=nug
  )
```

MAP estimates

```
1 with model:
2   gp_map = pm.find_MAP()
```

```
MAP                                          1% 0:00:16 logp = 189.21, ||grad|| = 0.091468
```

```
1 pprint(gp_map)
```

```
{'l': array(0.18938),
 'l_log__': array(-1.66401),
 'nug': array(0.09815),
 'nug_log__': array(-2.32123),
 's': array(0.37139),
 's_log__': array(-0.99049)}
```

Full Posterior Sampling

```
1 with model:
2   post_nuts = pm.sample(cores=2)
```

```
Progress                  Draws   Divergences   Step size   Grad evals   Sampling Speed   Elapsed
--------------------------------------------------------------------------------------------------
━━━━━━━━━━━━━━━━━━━━━━━━  1999    0             0.473       3            83.89 drawss/s   0:00:23
━━━━━━━━━━━━━━━━━━━━━━━━  1999    0             0.562       7            82.32 drawss/s   0:00:24
```

```
1 az.summary(post_nuts)
```

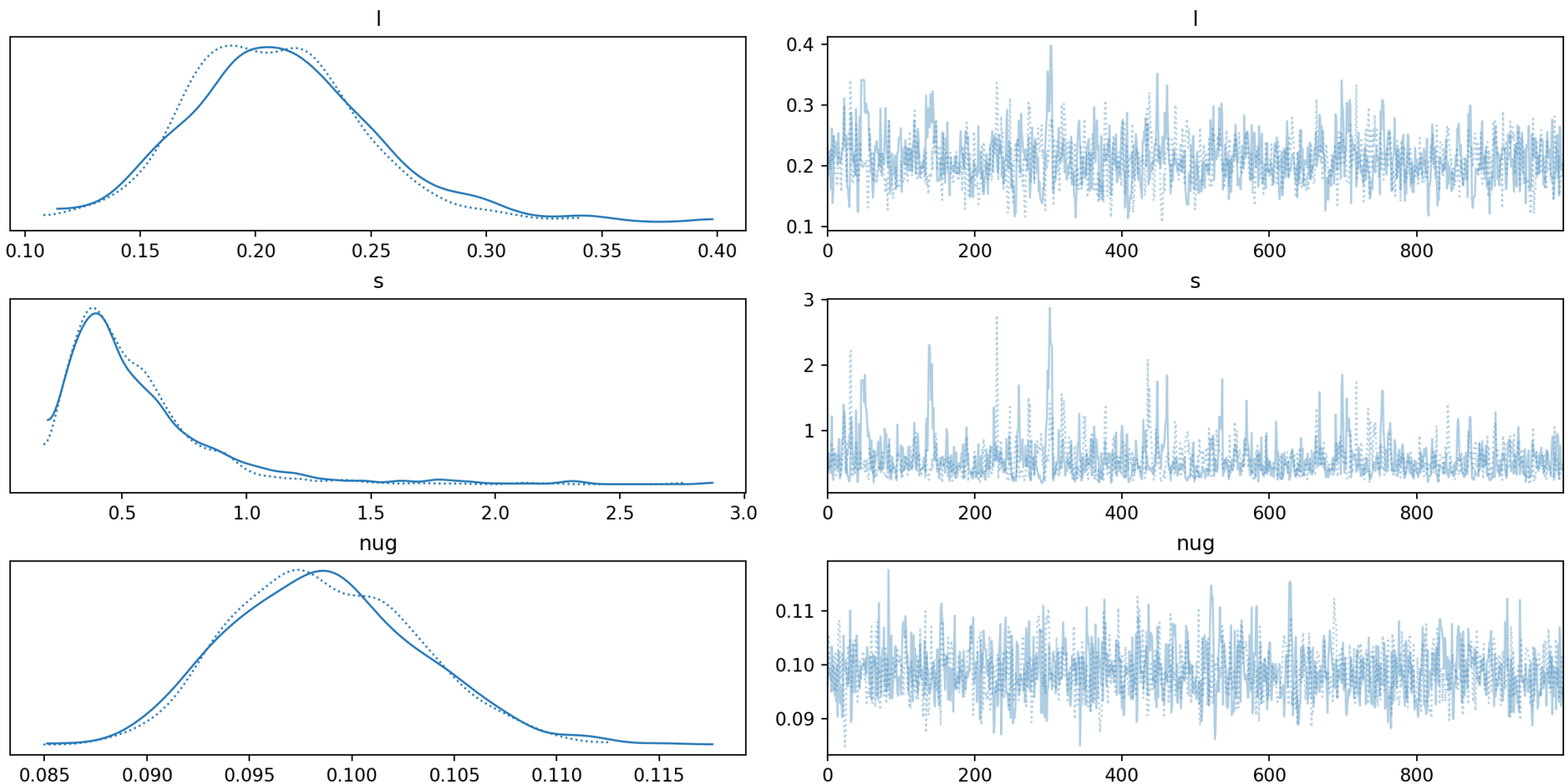
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.210	0.039	0.136	0.283	0.002	0.002	690.0	503.0	1.0
s	0.554	0.296	0.208	1.028	0.013	0.024	687.0	572.0	1.0
nug	0.099	0.005	0.090	0.107	0.000	0.000	1069.0	1041.0	1.0

```
1 # gp_map w/o logged parameters
2 {k: v for k, v in gp_map.items() if "log" not in k}
```

```
{'l': array(0.18938), 's': array(0.37139), 'nug': array(0.09815)}
```

Trace plots

```
1 ax = az.plot_trace(post_nuts)
2 plt.gcf().set_layout_engine("constrained")
3 plt.show()
```

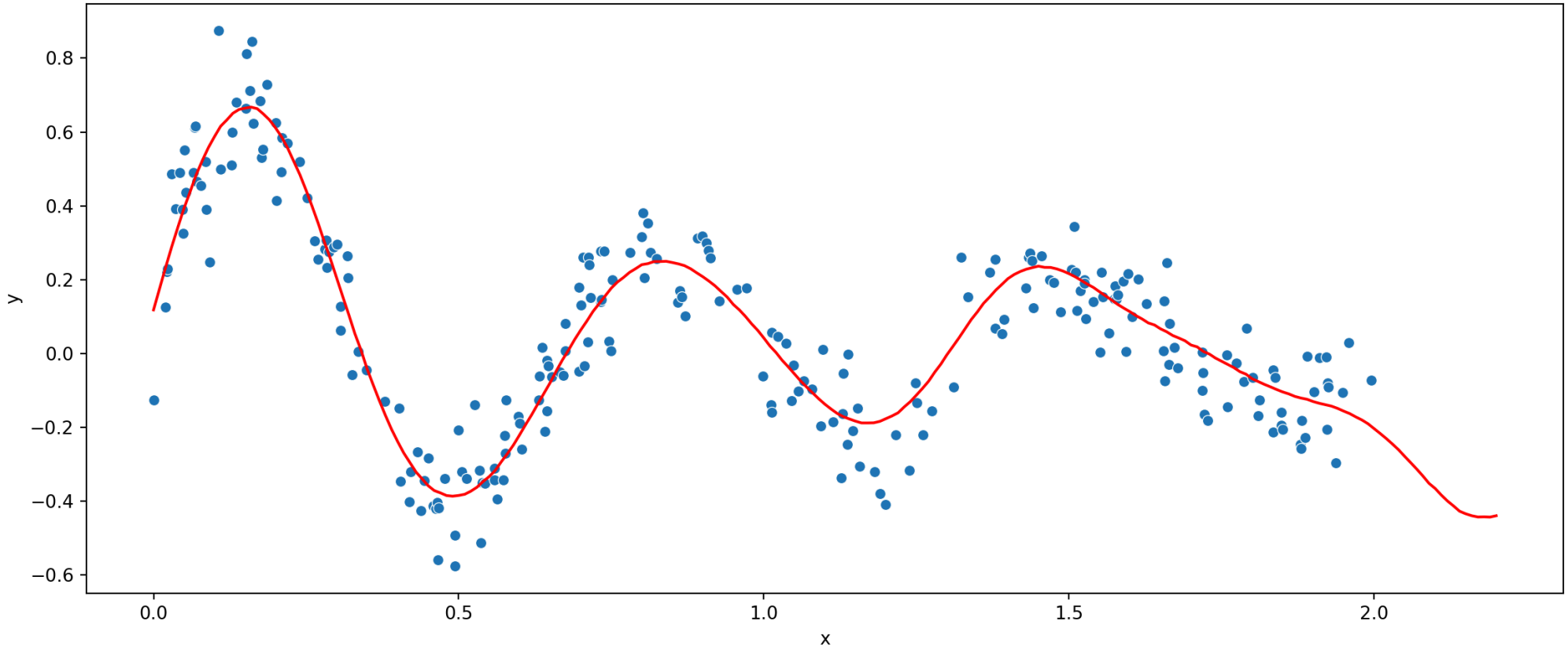


Conditional Predictions (MAP)

```
X_new = np.linspace(0, 2.2, 221).reshape(-1, 1)

with model:
  y_pred = gp.conditional("y_pred", X_new)
  pred_map = pm.sample_posterior_predictive(
    [gp_map], var_names=["y_pred"]
  )
```

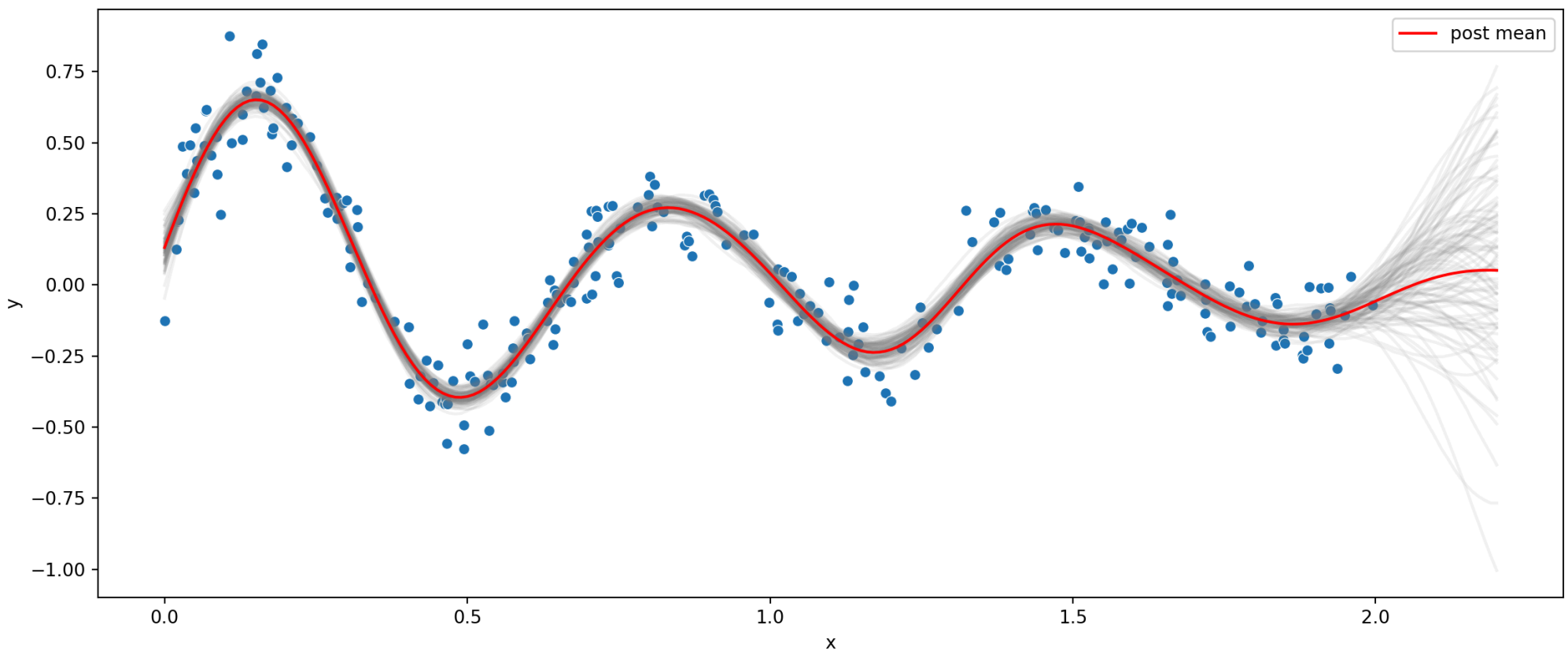
```
Sampling ... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00 / 0:00:00
```


0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
y
0.0
0.5
1.0
1.5
2.0
x

Conditional Predictions (full posterior)

```
with model:
  pred_post = pm.sample_posterior_predictive(
    post_nuts.sel(draw=slice(None,None,10)), var_names=["y_pred"]
  )
```

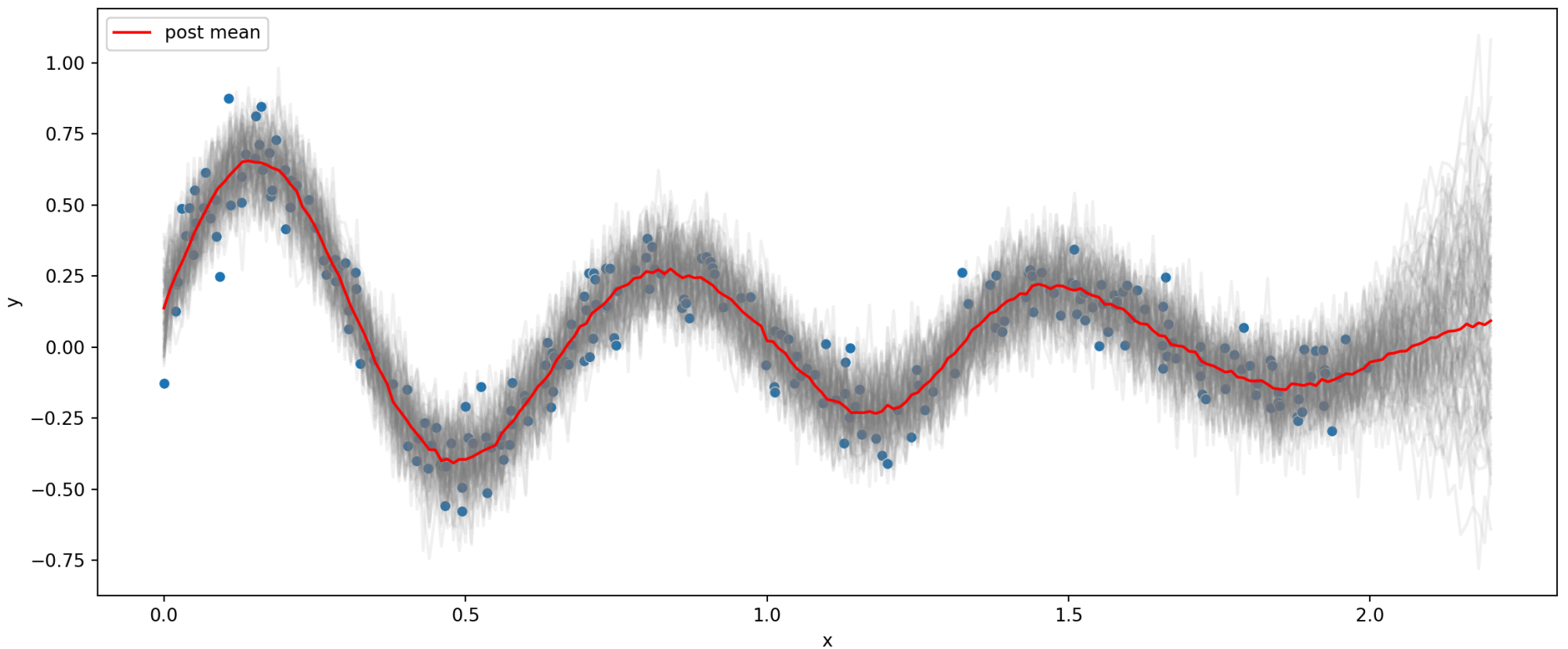
```
Sampling ... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00 / 0:00:00
```



Conditional Predictions (posterior + nugget)

```
with model:
  y_star = gp.conditional("y_star", X_new, pred_noise=True)
  predn_post = pm.sample_posterior_predictive(
    post_nuts.sel(draw=slice(None,None,10)), var_names=["y_star"]
  )
```

```
Sampling ... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00 / 0:00:00
```



Sampler Backends

Alternative NUTS sampler backends

Beyond the ability of PyMC to use different sampling steps - it can also use different sampler algorithm implementations to run your model.

These can be changed via the `nuts_sampler` argument which currently supports:

- `pymc` - standard NUTS sampler using PyMC's PyTensor backend
- `blackjax` - uses the blackjax library which is a collection of samplers written for JAX
- `numpyro` - probabilistic programming library inspired by Pyro, built using JAX
- `nutpie` - provides a wrapper to the `nuts-rs` Rust library (slight variation on NUTS implementation)

Notes on installation

Before using the above sampler backends, you will need to install the relevant packages.

For example, to use the `blackjax` sampler, you will need to install the `blackjax` package and its dependencies (e.g. `jax`). Similarly, for `numpyro`, you will need to install the `numpyro` package and its dependencies.

Many of these packages have extras that also need to be designated if you want full functionality (e.g. GPU support). Some common examples:

- `uv add "jax[cuda]"` to add CUDA support to JAX
- `uv add "nutpie[all]"` to add both pymc and stan support for nutpie

Sampler backend comparison

The four backends differ in their underlying implementation and parallelism model:

Backend	Language	Parallelism
`pymc`	C (Aesara/PyTensor)	1 core per chain
`blackjax`	JAX (XLA-compiled)	Multiple cores / GPU across chains
`numpyro`	JAX (XLA-compiled)	Multiple cores / GPU across chains
`nutpie`	Rust (`nuts-rs`)	1 core per chain

- JAX-based samplers (`blackjax`, `numpyro`) JIT-compile the model and can exploit multi-core CPUs or GPUs, but have higher compilation overhead on first run
- `pymc` and `nutpie` run each chain on a single core; chains are run in parallel via Python multiprocessing
- On small models the JAX compilation cost can dominate; on large models or with GPU hardware the JAX backends tend to win

The nutpie sampler

`nutpie` wraps the `nuts-rs` Rust implementation of NUTS:

- Written in Rust for low-overhead, cache-friendly execution — no Python/C interpreter overhead per leapfrog step
- Uses a **mass-matrix adaptation** scheme that estimates the full dense mass matrix (vs. PyMC's diagonal default), which can improve sampling efficiency on correlated posteriors
- Supports both PyMC models and Stan models
- Supports pre-compiling the model which separates the compilation cost from sampling

Performance

```
start = time.time()
with model:
    post_nuts = pm.sample(
      nuts_sampler="pymc", chains=4, progressbar
    )
print(f"pymc: {time.time() - start:.1f}s")
```

```
pymc: 31.3s
```

```
start = time.time()
with model:
    post_blackjax = pm.sample(
      nuts_sampler="blackjax", chains=4, progres
    )
print(f"blackjax: {time.time() - start:.1f}s")
```

```
blackjax: 27.8s
```

```
start = time.time()
with model:
    post_pyro = pm.sample(
      nuts_sampler="numpyro", chains=4, prog
    )
print(f"numpyro: {time.time() - start:.1f}s"
```

```
numpyro: 26.5s
```

```
start = time.time()
with model:
    post_nutpie = pm.sample(
      nuts_sampler="nutpie", chains=4, progr
    )
print(f"nutpie: {time.time() - start:.1f}s"
```

```
nutpie: 29.8s
```

```
import nutpie
compiled = nutpie.compile_pymc_model(model)
```

```
start = time.time()
post_nutpie2 = nutpie.sample(compiled, chains=4, progress_bar=False)
print(f"nutpie (compiled): {time.time() - start:.1f}s")
```

```
nutpie (compiled): 13.0s
```

Results

```
1 az.summary(post_nuts)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.212	0.041	0.137	0.286	0.001	0.001	1558.0	835.0	1.01
s	0.569	0.339	0.210	1.107	0.013	0.033	1453.0	916.0	1.01
nug	0.099	0.005	0.091	0.107	0.000	0.000	2050.0	1749.0	1.00

```
1 az.summary(post_blackjax)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.211	0.040	0.140	0.283	0.001	0.001	1599.0	1379.0	1.0
s	0.555	0.313	0.197	1.051	0.009	0.024	1696.0	1596.0	1.0
nug	0.099	0.005	0.090	0.108	0.000	0.000	2247.0	1707.0	1.0

```
1 az.summary(post_pyro)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.211	0.041	0.139	0.290	0.001	0.001	1603.0	1535.0	1.0
s	0.566	0.325	0.201	1.092	0.010	0.020	1512.0	1303.0	1.0
nug	0.099	0.005	0.091	0.108	0.000	0.000	2306.0	2134.0	1.0

```
1 az.summary(post_nutpie.posterior[["l","s","nug"]])
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.211	0.042	0.140	0.292	0.001	0.001	1150.0	1173.0	1.01
s	0.568	0.350	0.194	1.070	0.014	0.043	1226.0	1095.0	1.01
nug	0.099	0.005	0.090	0.107	0.000	0.000	2573.0	2511.0	1.00

```
1 az.summary(post_nutpie2.posterior[["l","s","nug"]])
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.211	0.040	0.143	0.290	0.001	0.001	1408.0	1439.0	1.0

Stan

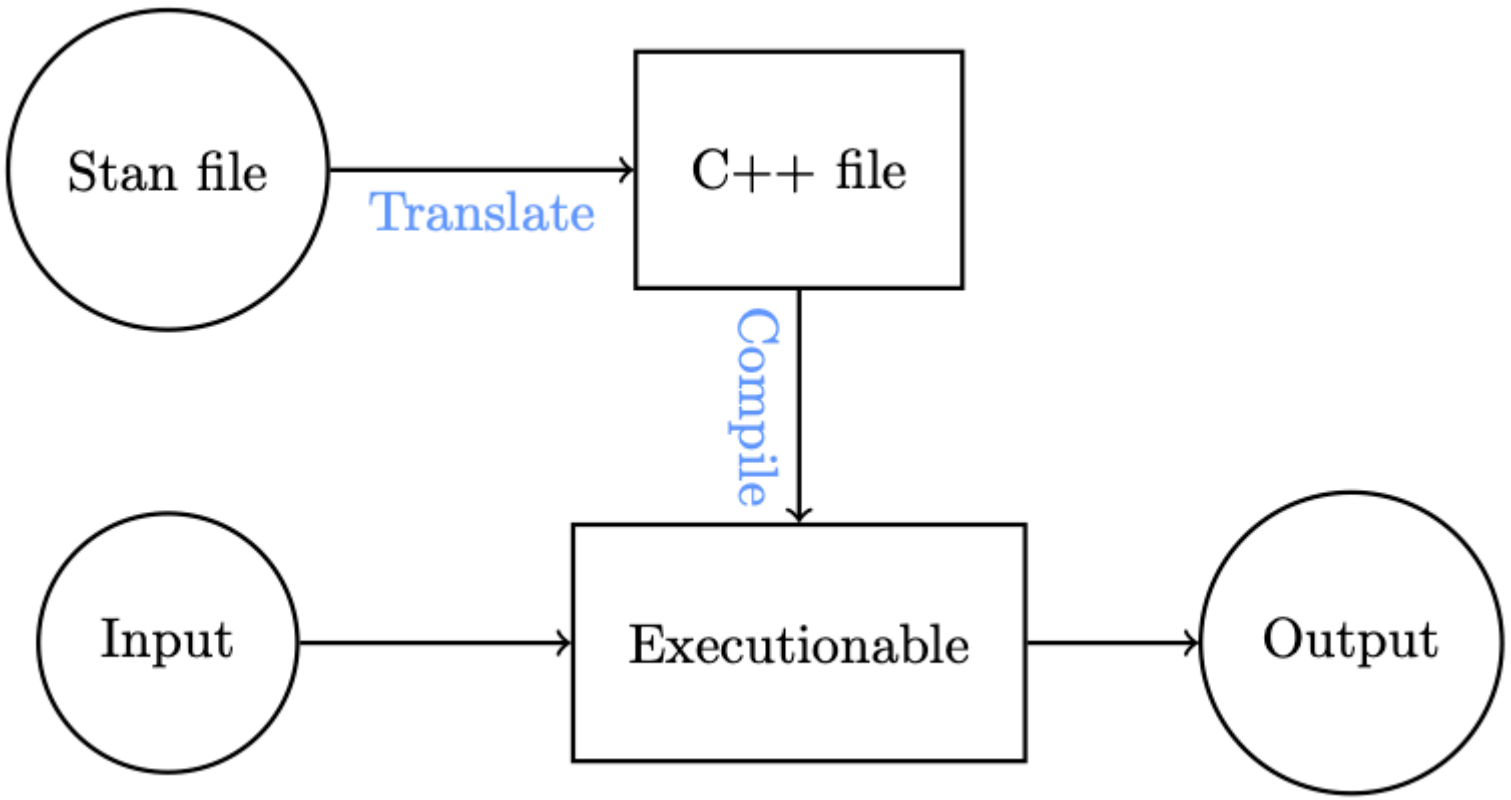
Stan in Python & R

At the moment both Python & R offer two variants of Stan:

- `pystan` & `RStan` - native language interface to the underlying Stan C++ libraries
  - The former does not play nicely with Jupyter (or quarto or positron) - see here for a fix
- `CmdStanPy` & `CmdStanR` - are wrappers around the `CmdStan` command line interface
  - Interface is through files (e.g. `./model.stan`)

Any of the above tools will require a modern C++ toolchain (C++17 support required).

Stan process



Stan file basics

Stan code is divided up into specific blocks depending on usage - all of the following blocks are optional but the ordering has to match what is given below.

```
functions {
  // user-defined functions
}
data {
  // declares the required data for the model
}
transformed data {
   // allows the definition of constants and transformations of the data
}
parameters {
   // declares the model's parameters
}
transformed parameters {
   // variables defined in terms of data and parameters
}
model {
   // defines the log probability function
}
generated quantities {
   // derived quantities based on parameters, data, and random number generation
}
```

GP model in Stan

Lec20/gp.stan

```
data {
  int<lower=1> N;
  array[N] real x;
  vector[N] y;
}
parameters {
  real<lower=0> l;
  real<lower=0> s;
  real<lower=0> nug;
}
model {
  // Covariance
  matrix[N, N] K = gp_exp_quad_cov(x, s, l);
  K = add_diag(K, nug^2);
  matrix[N, N] L = cholesky_decompose(K);

  // priors
  l ~ gamma(2, 1);
  s ~ cauchy(0, 5);
  nug ~ cauchy(0, 1);

  // model
```

Fit

```
from cmdstanpy import CmdStanModel
d_stan = d.to_dict('list')
d_stan["N"] = len(d["x"])
```

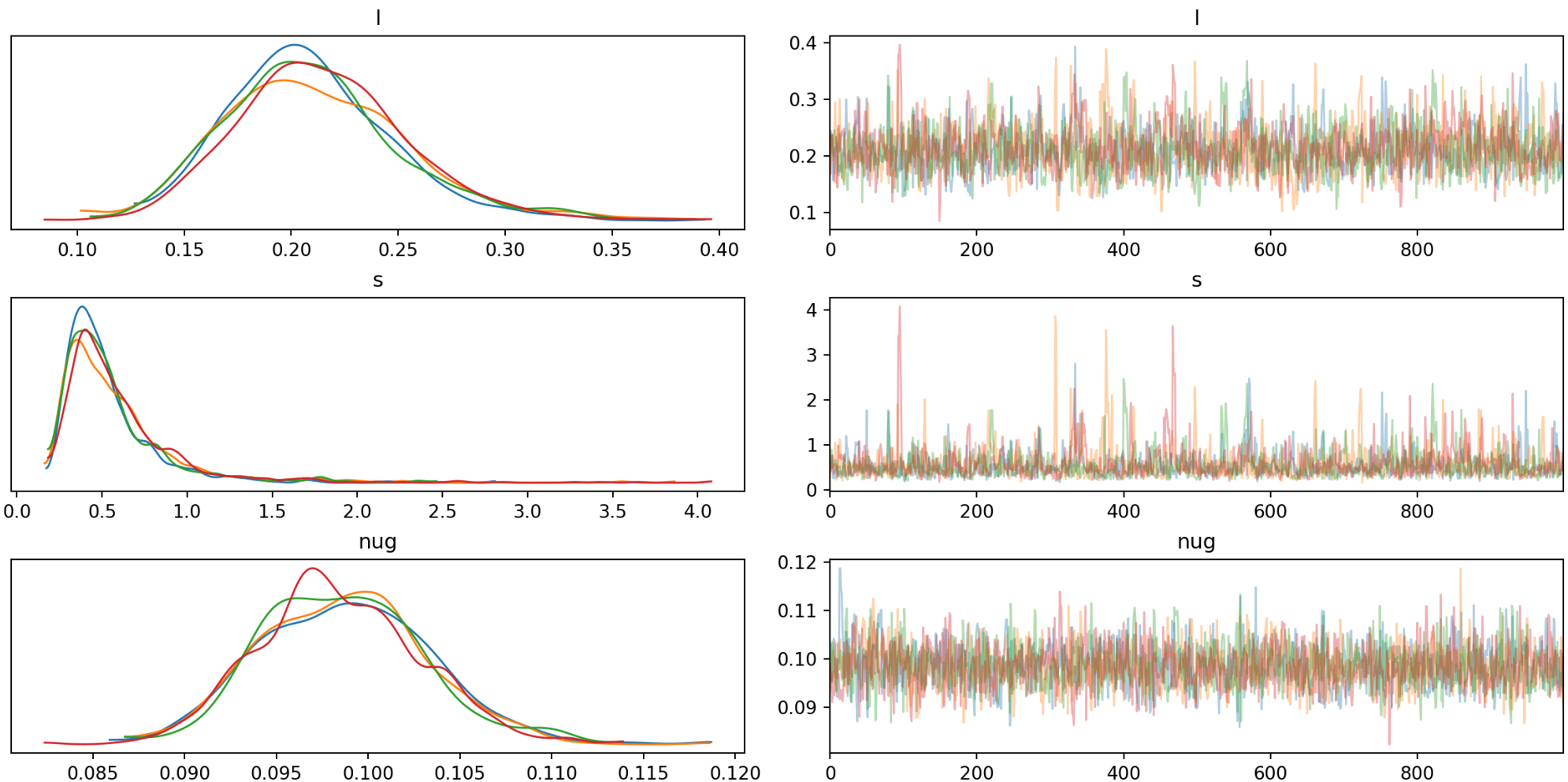
```
gp = CmdStanModel(stan_file='Lec20/gp.stan')
gp_fit = gp.sample(data=d_stan, show_progress=False)
```

```
gp_fit.summary()
```

	Mean	MCSE	StdDev	MAD	5%	50%	95%	ES
lp__	416.696000	0.031684	1.228810	1.015580	414.302000	416.996000	418.034000	1
l	0.211213	0.001178	0.041281	0.038427	0.150708	0.207788	0.283732	1
s	0.568497	0.010401	0.334768	0.198640	0.265488	0.481437	1.158120	1
nug	0.098575	0.000104	0.004480	0.004533	0.091598	0.098449	0.106188	1

Trace plots

```
1 ax = az.plot_trace(gp_fit, compact=False)
2 plt.show()
```



Diagnostics

```
1 gp_fit.divergences
```

```
array([0, 0, 0, 0])
```

```
1 gp_fit.max_treedepths
```

```
array([0, 0, 0, 0])
```

```
1 gp_fit.method_variables().keys()
```

```
dict_keys(['lp__', 'accept_stat__', 'stepsize__', 'treedepth__', 'n_leapfrog__', 'divergent__', 'e
```

```
1 print(gp_fit.diagnose())
```

```
Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
No divergent transitions found.

Checking E-BFMI - sampler transitions HMC potential energy.
E-BFMI satisfactory.

Rank-normalized split effective sample size satisfactory for all parameters.

Rank-normalized split R-hat values satisfactory for all parameters.

Processing complete, no problems detected.
```

nutpie & stan

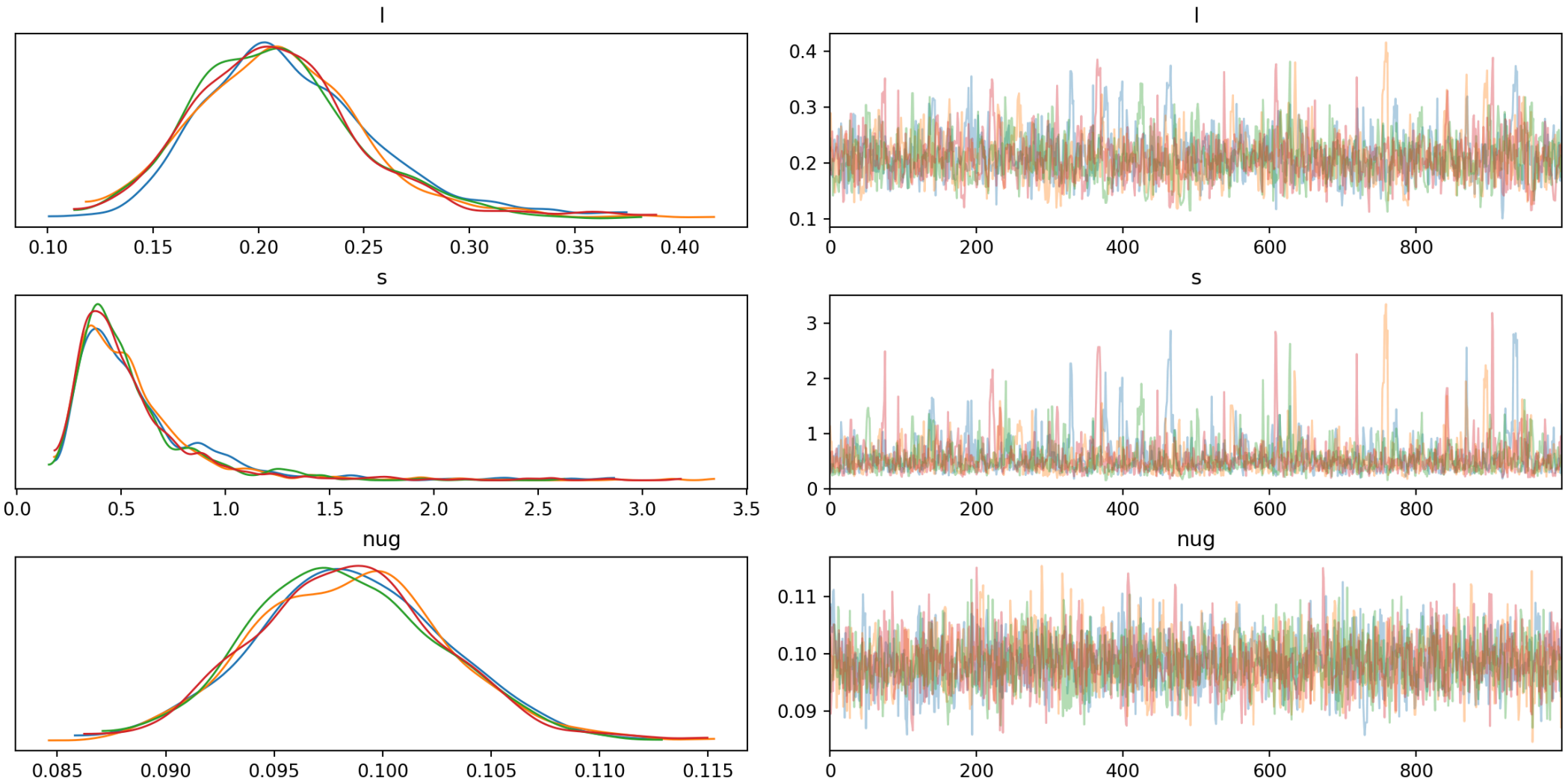
The `nutpie` package can also be used to compile and run stan models, it uses a package called `bridgestan` to interface with Stan.

```
import nutpie
m = nutpie.compile_stan_model(filename="Lec20/gp.stan")
m = m.with_data(x=d["x"],y=d["y"],N=len(d["x"]))
gp_fit_nutpie = nutpie.sample(m, chains=4)
```

```
az.summary(gp_fit_nutpie)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.211	0.042	0.140	0.292	0.001	0.001	1342.0	1219.0	1.0
s	0.564	0.333	0.205	1.100	0.011	0.022	1359.0	1136.0	1.0
nug	0.098	0.004	0.090	0.107	0.000	0.000	2353.0	2351.0	1.0

Trace plots


l
0.10
0.15
0.20
0.25
0.30
0.35
0.40
l
0.4
0.3
0.2
0.1
0
200
400
600
800
s
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
s
3
2
1
0
0
200
400
600
800
nug
0.085
0.090
0.095
0.100
0.105
0.110
0.115
nug
0.11
0.10
0.09
0
200
400
600
800

Performance

```
t_stan_gp = timeit.repeat(
    lambda: gp.sample(data=d_stan, show_progress=False),
    repeat=3, number=1
)
```

```
34.81s ± 0.68s (3 runs, 1 loop each)
```

```
t_stan_nutpie_gp = timeit.repeat(
    lambda: nutpie.sample(m, chains=4, progress_bar=False),
    repeat=3, number=1
)
```

```
14.83s ± 0.59s (3 runs, 1 loop each)
```

Posterior predictive model

Lec20/gp2.stan

```
functions {
  // From https://mc-stan.org/docs/stan-users-guide/gaussian-processes.html#predictive-inference-with-a
  vector gp_pred_rng(
    array[] real x2,
    vector y1,
    array[] real x1,
    real alpha,
    real rho,
    real sigma,
    real delta
  ) {
    int N1 = rows(y1);
    int N2 = size(x2);
    vector[N2] f2;
    {
      matrix[N1, N1] L_K;
      vector[N1] K_div_y1;
      matrix[N1, N2] k_x1_x2;
      matrix[N1, N2] v_pred;
      vector[N2] f2_mu;
      matrix[N2, N2] cov_f2;
      matrix[N2, N2] diag_delta;
      matrix[N1, N1] K;
      K = gp_exp_quad_cov(x1, alpha, rho);
```

Posterior predictive fit

```
1 d_stan = d.to_dict('list')
2 d_stan["N"] = len(d_stan["x"])
3 d_stan["xp"] = np.linspace(0, 2.2, 221)
4 d_stan["Np"] = len(d_stan["xp"])
```

```
1 gp2 = CmdStanModel(stan_file='Lec20/gp2.stan')
2 gp2_fit = gp2.sample(data=d_stan, show_progress=False)
```

```
1 gp2_fit.summary()
```

	Mean	MCSE	StdDev	MAD	5%	50%	95%
lp__	416.632000	0.035060	1.311560	1.051900	414.023000	416.983000	418.035000
l	0.211678	0.000984	0.040204	0.037695	0.150728	0.208051	0.285263
s	0.564591	0.007880	0.311079	0.196101	0.268283	0.481254	1.135830
nug	0.098698	0.000110	0.004787	0.004621	0.091180	0.098506	0.107040
f[1]	0.134059	0.000817	0.048615	0.049339	0.052479	0.135168	0.212925
...	...	...	...	...	...	...	...
f[217]	0.089114	0.004508	0.281242	0.268842	−0.350164	0.083123	0.573051
f[218]	0.093848	0.004769	0.297292	0.282442	−0.372340	0.085499	0.605130
f[219]	0.097935	0.005028	0.313258	0.296261	−0.393446	0.088212	0.643016
f[220]	0.101391	0.005284	0.329082	0.310729	−0.412781	0.090177	0.666344
f[221]	0.104235	0.005536	0.344706	0.321935	−0.429439	0.091224	0.689960

[225 rows x 10 columns]

Draws

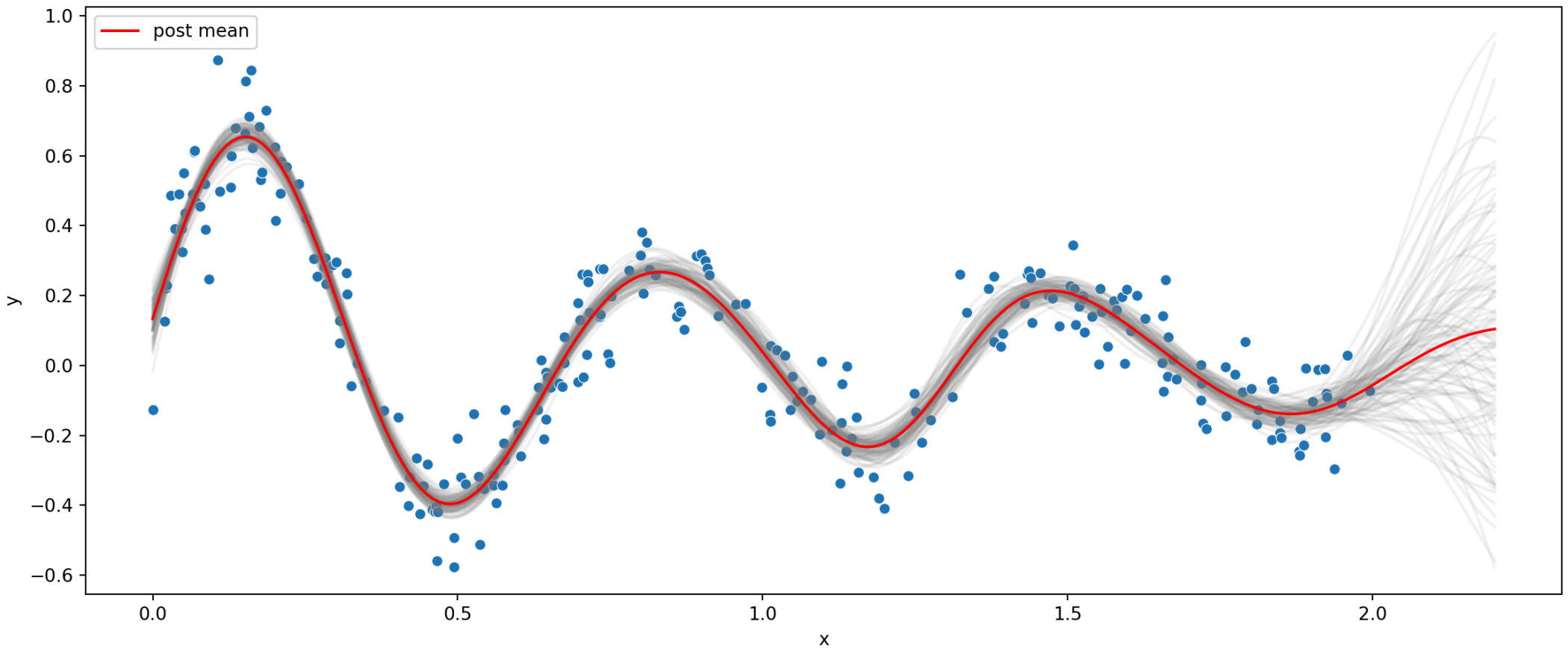
```
1 gp2_fit.stan_variable("f").shape
```

```
(4000, 221)
```

```
1 np.mean(gp2_fit.stan_variable("f"), axis=0)
```

```
array([ 0.13406,  0.19034,  0.24532,  0.29852,  0.34948,  0.39774,  0.44286,  0.48445,  0.52213,
        0.6173 ,  0.59586,  0.56997,  0.53995,  0.5061 ,  0.4688 ,  0.42842,  0.38539,  0.34013,
       -0.1767 , -0.21446, -0.24917, -0.28057, -0.30847, -0.33267, -0.35307, -0.36957, -0.38213, -
       -0.28403, -0.25857, -0.23149, -0.20309, -0.17367, -0.14353, -0.11296, -0.08225, -0.05166, -
        0.21452,  0.2284 ,  0.24019,  0.24987,  0.25743,  0.26288,  0.26622,  0.26749,  0.26674,
        0.14299,  0.12354,  0.10321,  0.08212,  0.06042,  0.03825,  0.01577, -0.00685, -0.02943, -
       -0.21965, -0.22654, -0.23096, -0.23283, -0.23208, -0.22867, -0.22262, -0.21398, -0.20282, -
        0.02721,  0.05058,  0.07306,  0.09442,  0.11442,  0.13288,  0.14963,  0.16454,  0.17754,
        0.19641,  0.18895,  0.18047,  0.17107,  0.16089,  0.15002,  0.13856,  0.12662,  0.11428,
       -0.02806, -0.04004, -0.0516 , -0.06269, -0.07325, -0.08323, -0.09256, -0.10118, -0.10903, -
       -0.13276, -0.12872, -0.1237 , -0.11775, -0.11094, -0.10332, -0.09497, -0.08598, -0.07643, -
        0.03826,  0.04728,  0.05577,  0.06369,  0.071  ,  0.07769,  0.08373,  0.08911,  0.09385,
```

Plot



Tennis Model Performance

The Data

- Jeff Sackmann's ATP tennis dataset — match records from 1968 to present
- Each match record contains winning and losing player names; these are encoded as integer IDs
- Goal is to estimate latent player skill levels from match outcomes using a hierarchical Bradley-Terry model
  - $\text{logit}\, P(\text{player } i \text{ beats player } j) = \text{skill}_i - \text{skill}_j$
- Benchmark datasets created by filtering from different `start_year` values (1970, 1980, 1990, 2000, 2010, 2020), giving datasets of increasing size

Model and data from Martin Ingram's "MCMC for big datasets – faster sampling with JAX and the GPU"; results based on

Testing Setup

All benchmarks run on a departmental server with no CPU or GPU constraints:

- *CPU* - AMD Ryzen 9 7950X (16-core / 32-thread)
- *GPU* - 2× NVIDIA RTX A4000 (16 GB VRAM each)
- Each sampler run with *2 chains*, *1000 draws*, *1000 tuning steps*
- JAX-based samplers (`numpyro`, `blackjax`) benchmarked in both `cpu_parallel` and `gpu_parallel` modes
- `nutpie` benchmarked with both PyMC and Stan compiled models
- Timing measured as wall-clock seconds per sampling run (including compile time)
- All CPU and GPU results are using 64-bit precision

The Models

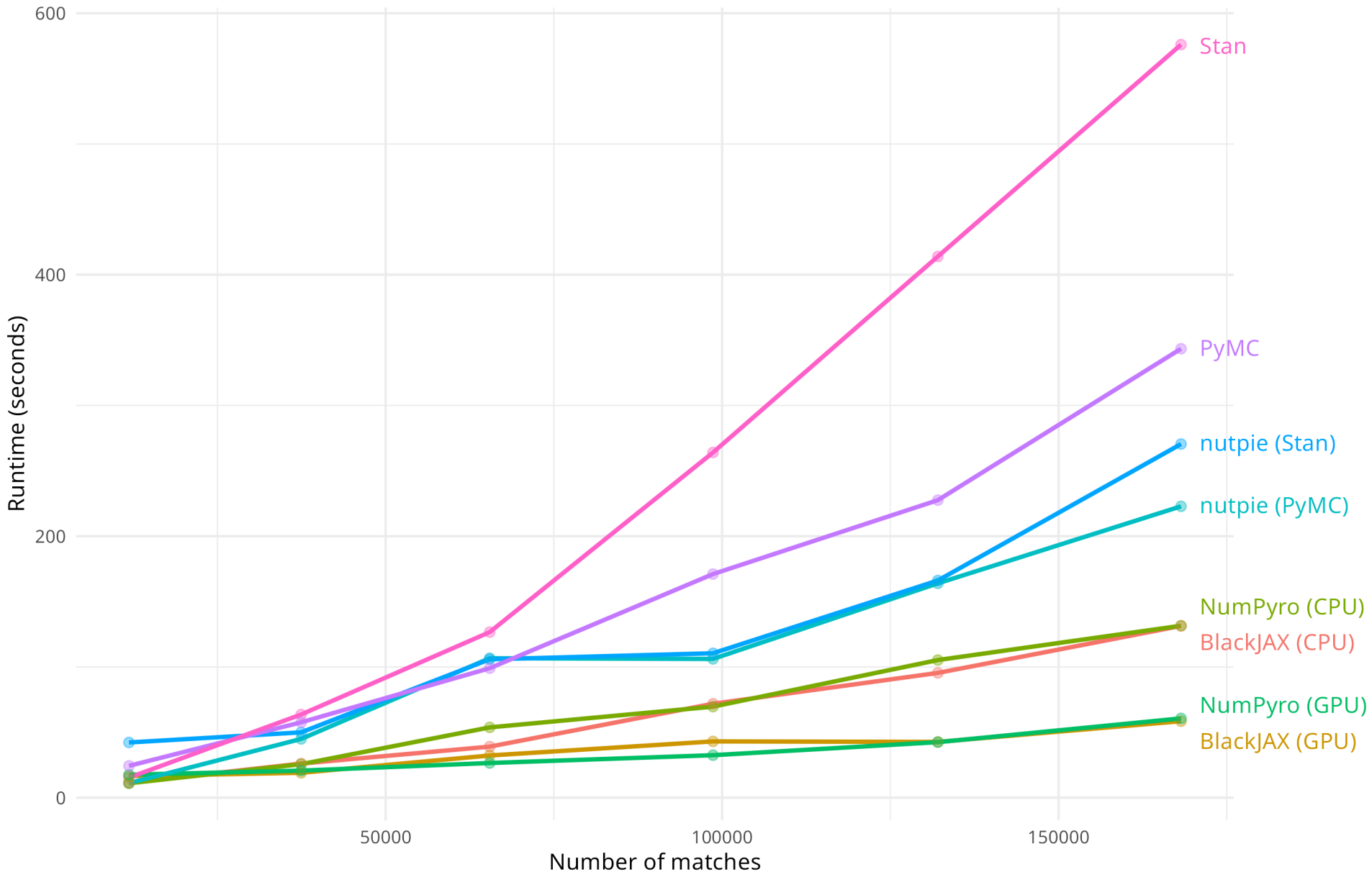
PyMC

```
with pm.Model() as model:
    player_sd = pm.HalfNormal("player_sd", s
    player_skills_raw = pm.Normal(
        "player_skills_raw", 0., sigma=1.,
        shape=(n_players,)
    )
    player_skills = pm.Deterministic(
        "player_skills", player_skills_raw *
    )
    logit_p = player_skills[winner_ids] - pl
    win_lik = pm.Bernoulli(
        "win_lik", logit_p=logit_p,
        observed=np.ones(n_matches)
    )
```

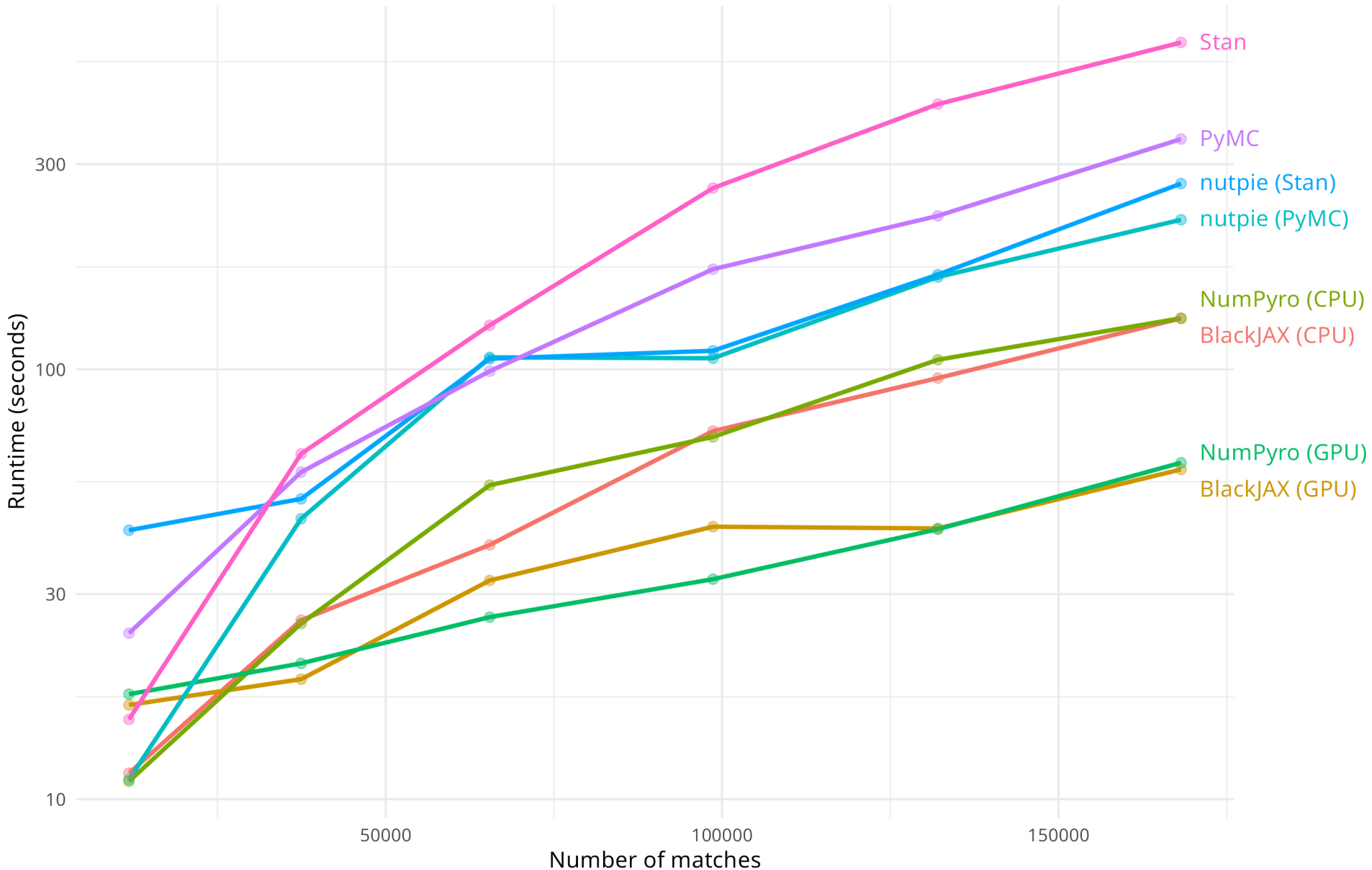
Stan

```
parameters {
    vector[n_players] player_skills_raw;
    real<lower=0> player_sd;
}
transformed parameters {
    vector[n_players] player_skills =
        player_skills_raw * player_sd;
}
model {
    player_skills_raw ~ std_normal();
    player_sd ~ normal(0, 1);
    vector[n_matches] mu;
    for (n in 1:n_matches)
        mu[n] = player_skills[winner_ids[n]]
              - player_skills[loser_ids[n]];
    1 ~ bernoulli_logit(mu);
}
```

Results


600
400
200
0
Runtime (seconds)
50000
100000
150000
Number of matches
Stan
PyMC
nutpie (Stan)
nutpie (PyMC)
NumPyro (CPU)
BlackJAX (CPU)
NumPyro (GPU)
BlackJAX (GPU)

Results (log scale)



Results (relative to NumPyro GPU)