MCMC - Performance & Stan

Lecture 25

Dr. Colin Rundel

Example - Gaussian Process

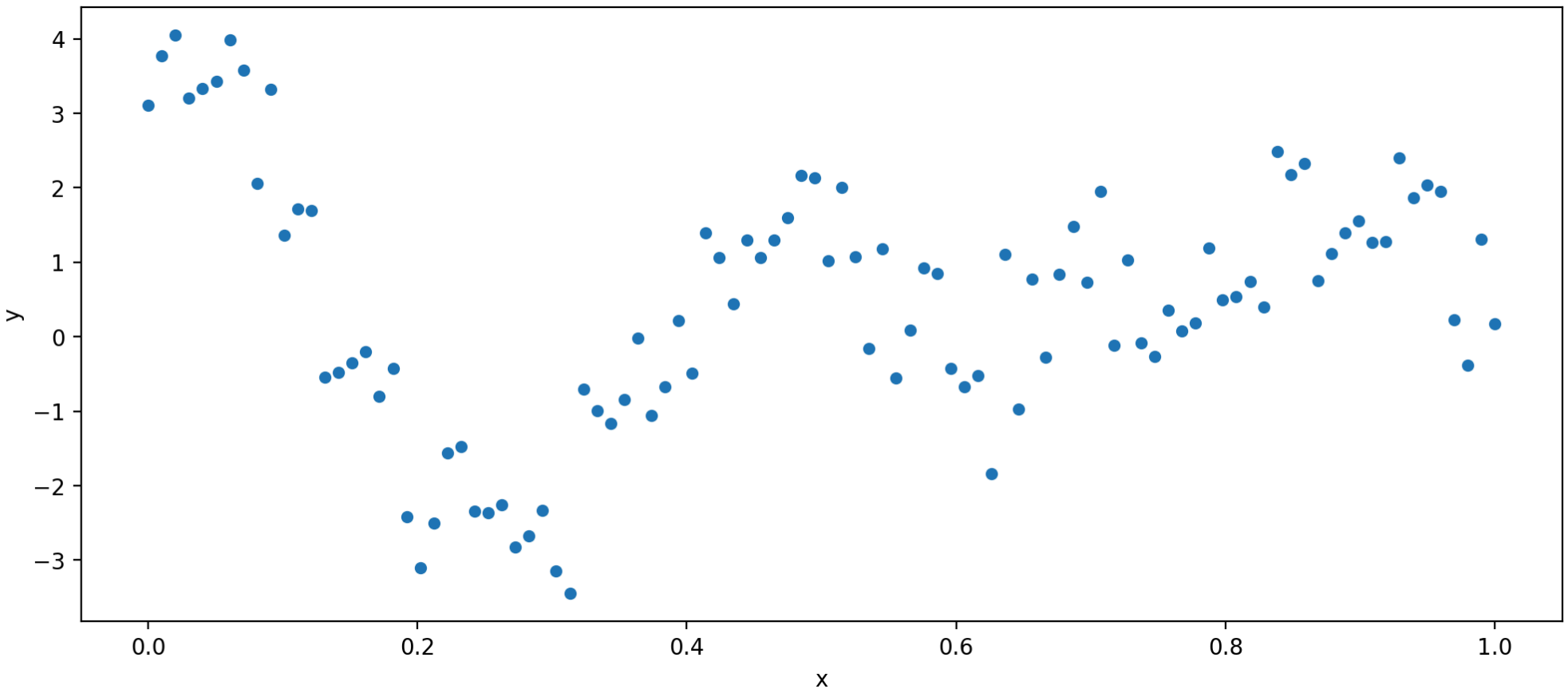
Data

```
1 d = pd.read_csv("data/gp2.csv")
2 d
```

	x	y
0	0.000000	3.113179
1	0.010101	3.774512
2	0.020202	4.045562
3	0.030303	3.207971
4	0.040404	3.336638
...	...	...
95	0.959596	1.951793
96	0.969697	0.224769
97	0.979798	-0.387220
98	0.989899	1.304032
99	1.000000	0.174600

100 rows × 2 columns

```
fig = plt.figure(figsize=(12, 5))
ax = sns.scatterplot(x="x", y="y", data=d)
plt.show()
```

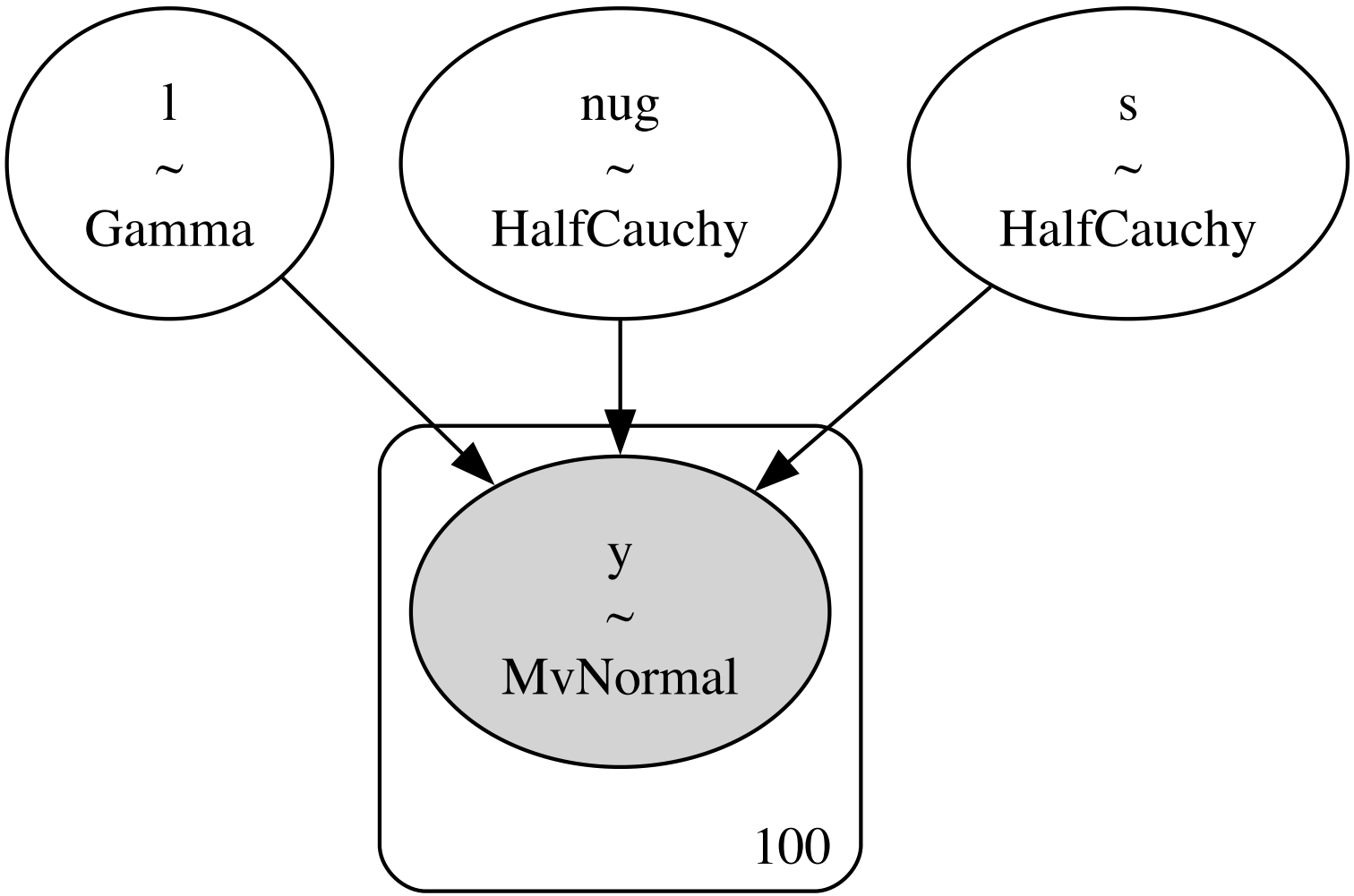


GP model

```
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)

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

Model visualization

```
1 pm.model_to_graphviz(model)
```



MAP estimates

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

```
MAP ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━  0% 0:00:04 logp = -134.97, ||grad|| = 0.0022532
```

```
1 gp_map
```

```
{'l_log__': array(-2.35319),
 's_log__': array(0.54918),
 'nug_log__': array(-0.33237),
 'l': array(0.09507),
 's': array(1.73184),
 'nug': array(0.71722)}
```

Full Posterior Sampling

```
with model:
  post_nuts = pm.sample()
```

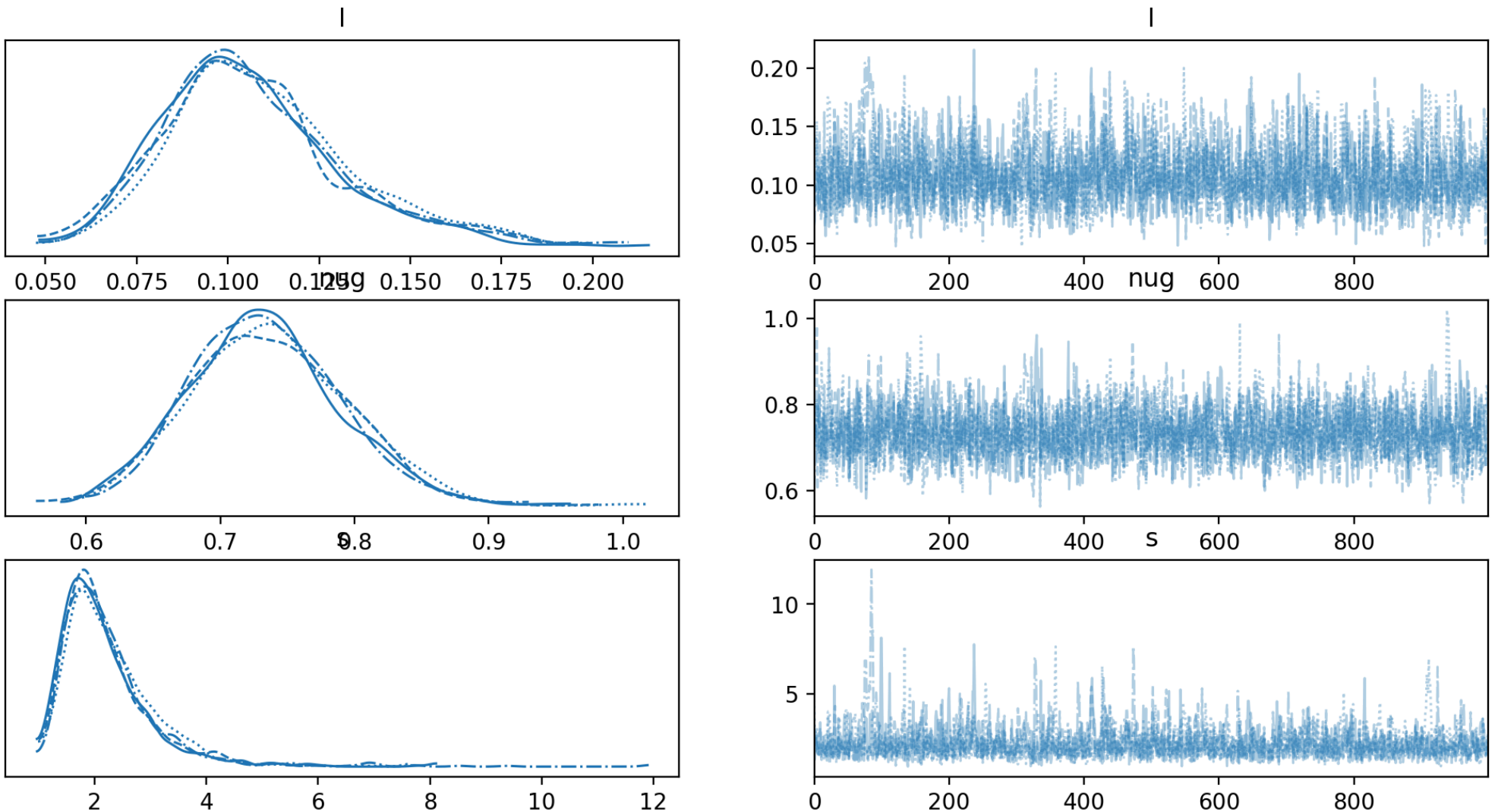
Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elap
————————————	2000	0	0.72	7	376.79 draws/s	0:00
————————————	2000	0	0.70	7	358.24 draws/s	0:00
————————————	2000	0	0.68	3	382.31 draws/s	0:00
————————————	2000	0	0.78	3	378.42 draws/s	0:00

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.107	0.025	0.064	0.159	0.001	0.001	1421.0	1564.0	1.0
nug	0.735	0.058	0.629	0.844	0.001	0.001	2336.0	1898.0	1.0
s	2.223	0.846	1.055	3.626	0.026	0.054	1580.0	1251.0	1.0

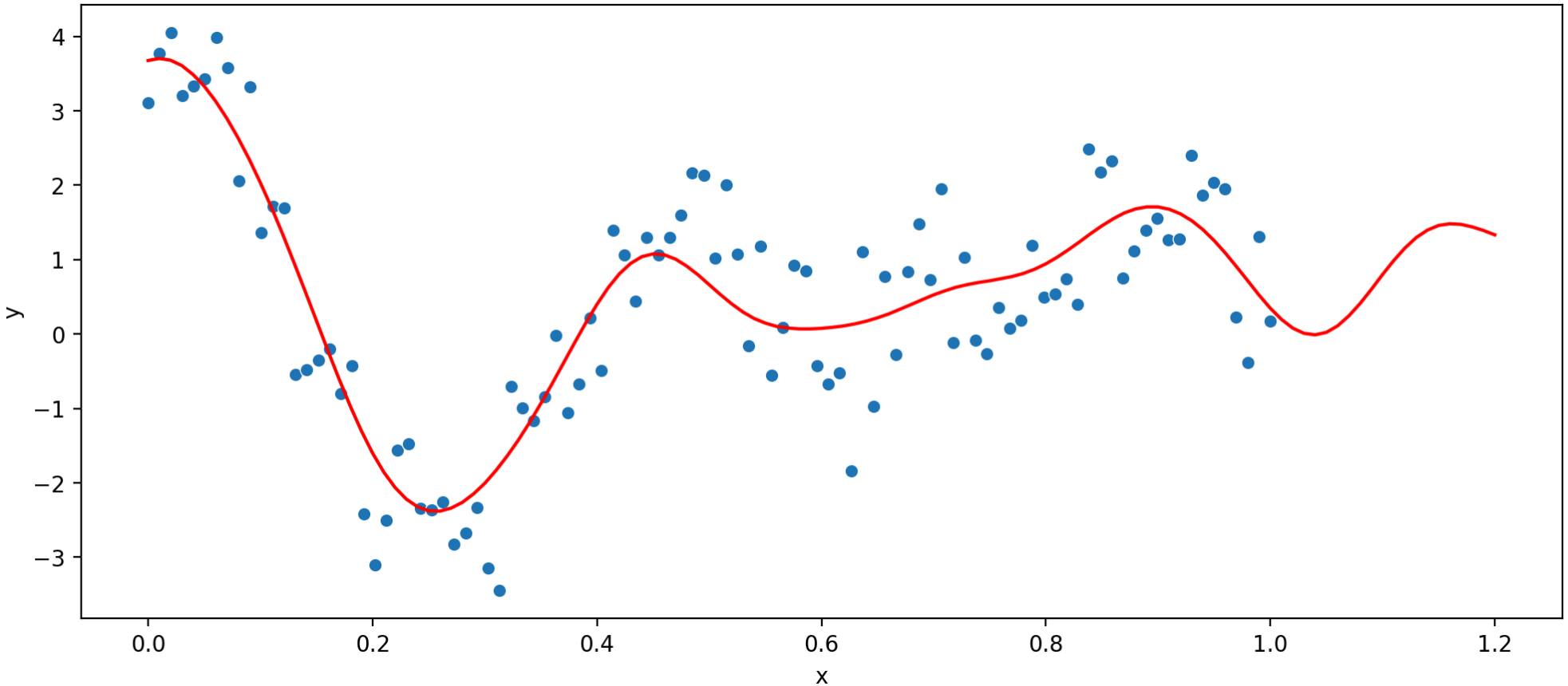
Trace plots

```
1  ax = az.plot_trace(post_nuts)
2  plt.show()
```



Conditional Predictions (MAP)

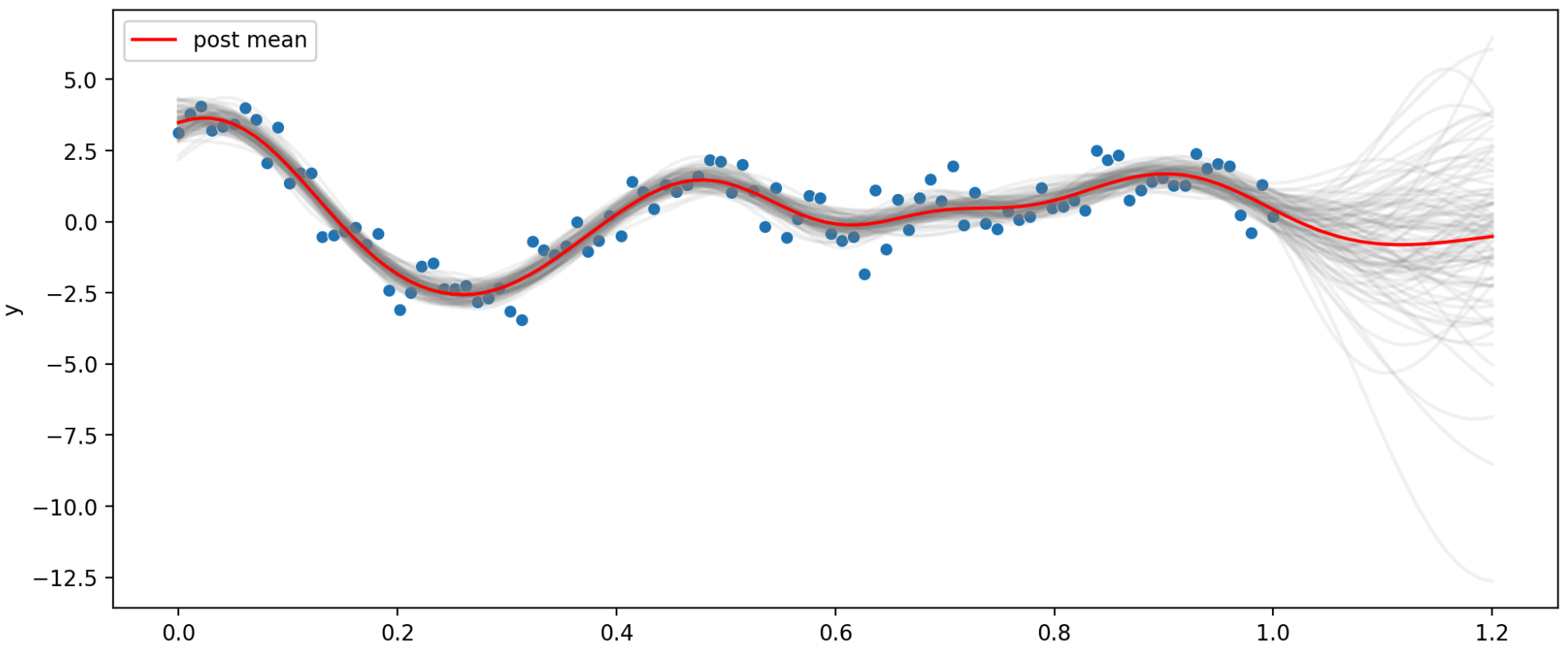
```
X_new = np.linspace(0, 1.2, 121).reshape(-1, 1)

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


4
3
2
1
0
−1
−2
−3
y
0.0
0.2
0.4
0.6
0.8
1.0
1.2
x

Conditional Predictions (thinned)

```
1 with model:
2   pred_post = pm.sample_posterior_predictive(
3     post_nuts.sel(draw=slice(None,None,10)), var_names=["y_pred"]
4   )
```

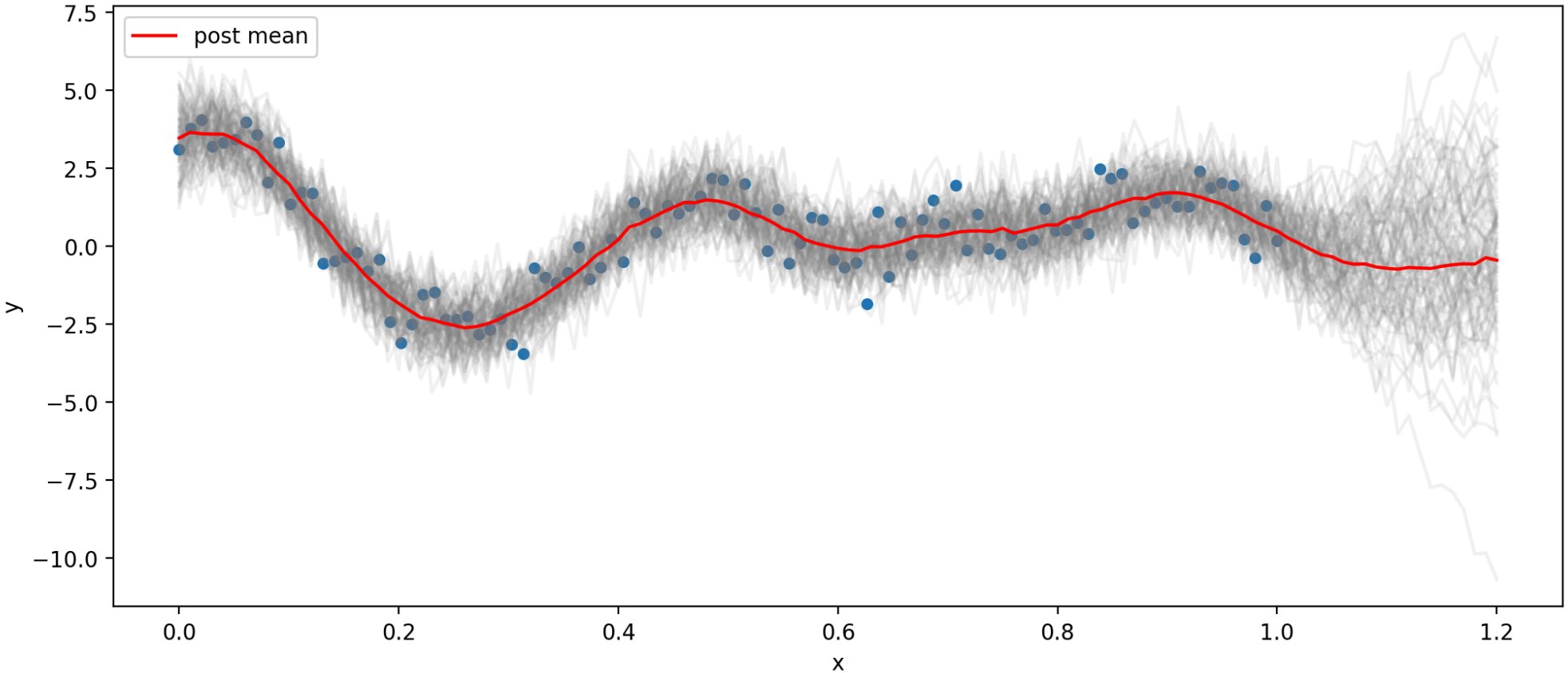
```
Sampling ... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00 / 0:00:00
```



Conditional Predictions w/ 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
```


post mean
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
y
0.0
0.2
0.4
0.6
0.8
1.0
1.2
x

Alternative NUTS samplers

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 sampler uses pymc's C backend
- `blackjax` - uses the blackjax library which is a collection of samplers written for JAX
- `numpyro` - probabilistic programming library for pyro built using JAX
- `nutpie` - provides a wrapper to the `nuts-rs` Rust library (slight variation on NUTS compared to numpy & stan)

Performance

```
%%timeit -r 3
with model:
  post_nuts = pm.sample(nuts_sampler="pymc", chains=4, progressbar=False)
```

```
6.26 s ± 83.3 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

```
%%timeit -r 3
with model:
    post_jax = pm.sample(nuts_sampler="blackjax", chains=4, progressbar=False)
```

```
4.16 s ± 120 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

```
%%timeit -r 3
with model:
    post_numpyro = pm.sample(nuts_sampler="numpyro", chains=4, progressbar=False)
```

```
3.74 s ± 72.4 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

```
%%timeit -r 3
with model:
    post_nutpie = pm.sample(nuts_sampler="nutpie", chains=4, progressbar=False)
```

```
9.98 s ± 216 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

Nutpie and compilation

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

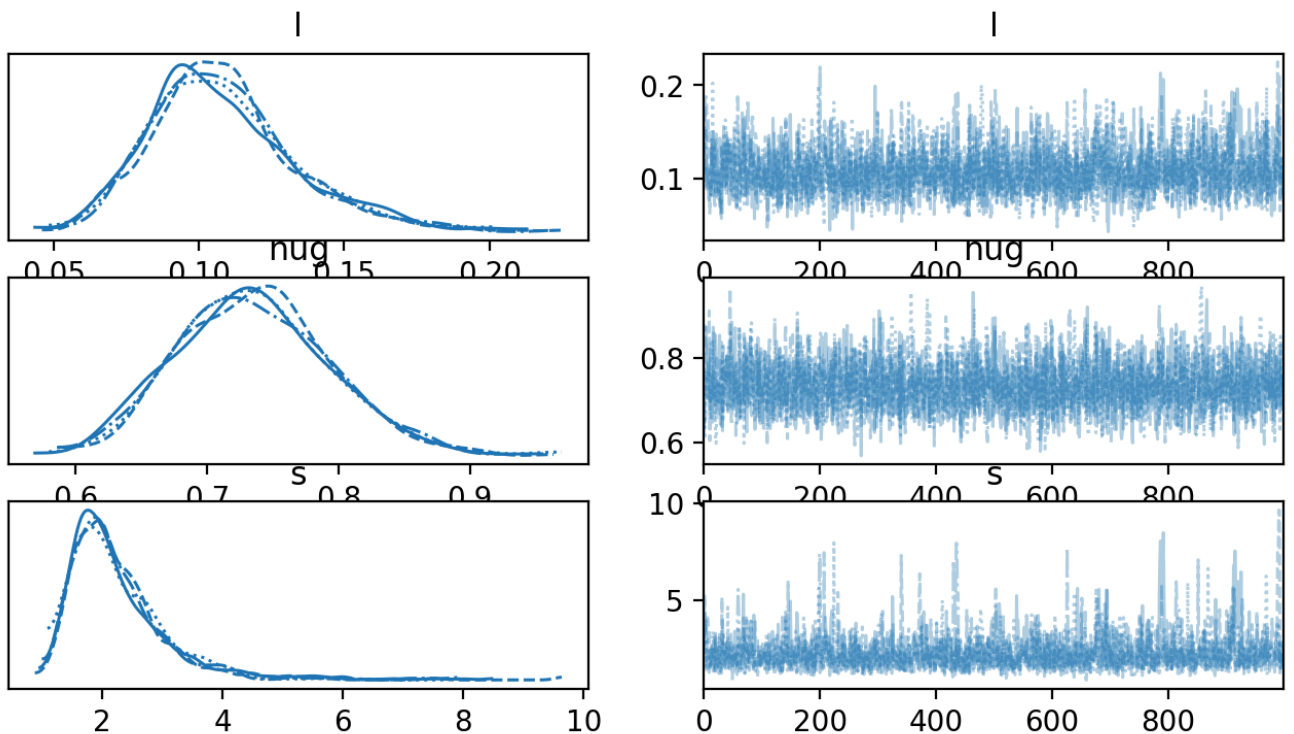
```
1 %%timeit -r 3
2 post_nutpie = nutpie.sample(compiled,chains=4, progress_bar=False)
```

```
2.92 s ± 64.3 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

JAX

```
1 az.summary(post_jax)
```

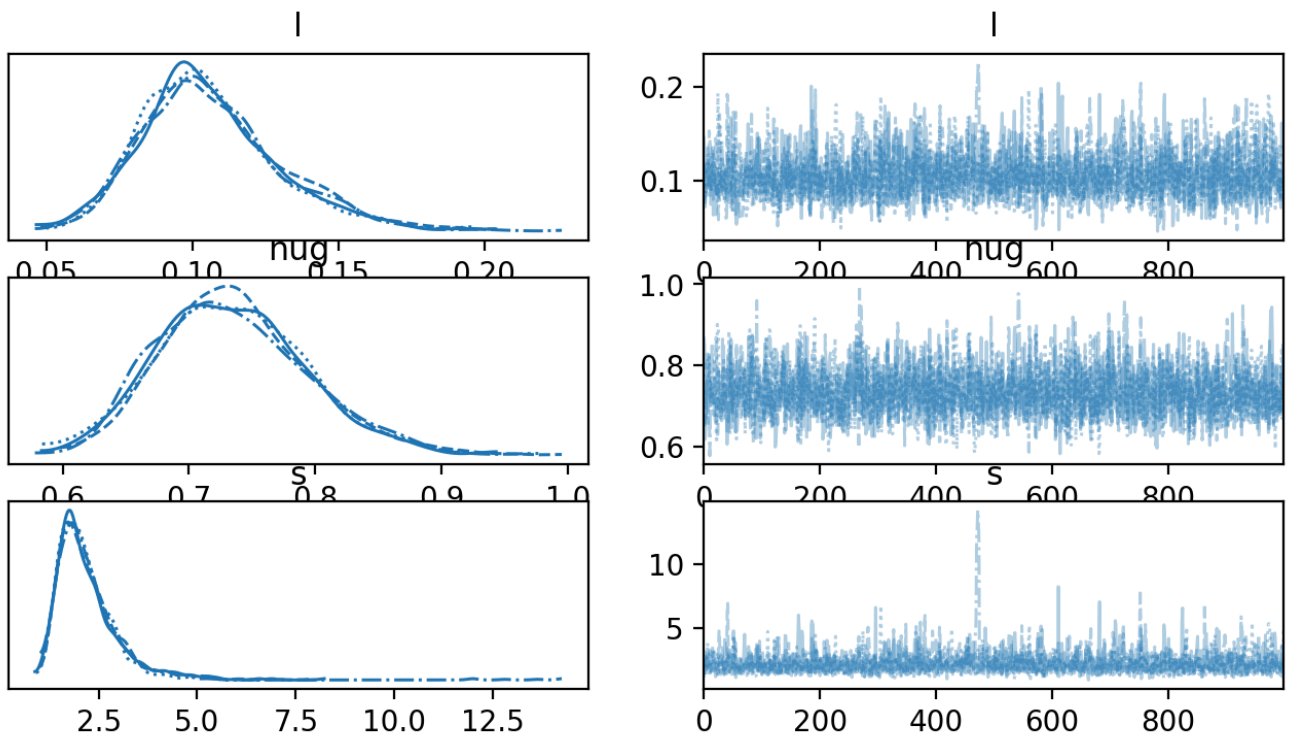
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.108	0.025	0.063	0.156	0.001	0.001	2184.0	1518.0	1.0
nug	0.736	0.058	0.623	0.838	0.001	0.001	2728.0	2894.0	1.0
s	2.266	0.875	1.126	3.747	0.025	0.048	1794.0	1567.0	1.0



Numpyro NUTS sampler

```
1 az.summary(post_numpyro)
```

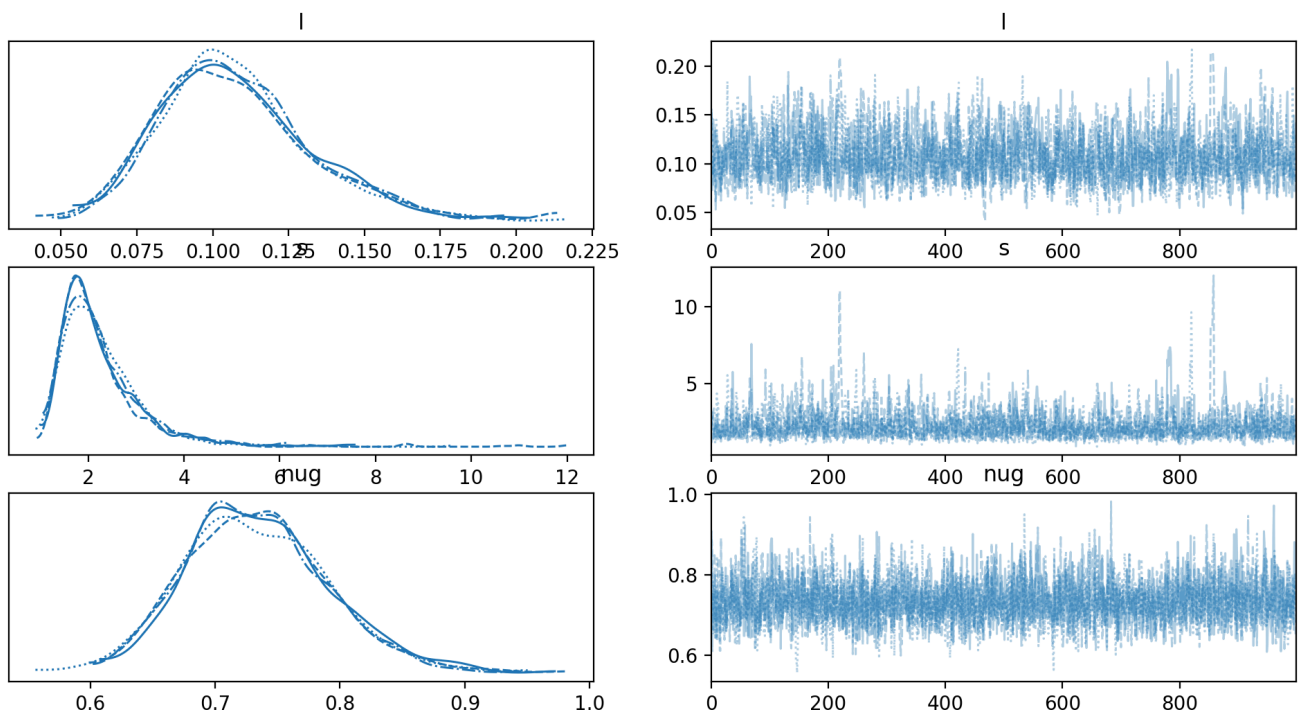
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.106	0.024	0.064	0.154	0.001	0.001	2078.0	2104.0	1.0
nug	0.735	0.059	0.630	0.851	0.001	0.001	2341.0	2340.0	1.0
s	2.208	0.860	1.084	3.572	0.023	0.088	2406.0	2067.0	1.0



nutpie sampler

```
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	0.108	0.025	0.061	0.154	0.001	0.001	1657.0	1905.0	1.0
s	2.251	0.908	1.045	3.737	0.026	0.061	1803.0	1642.0	1.0
nug	0.734	0.056	0.633	0.840	0.001	0.001	3375.0	2756.0	1.0



Stan

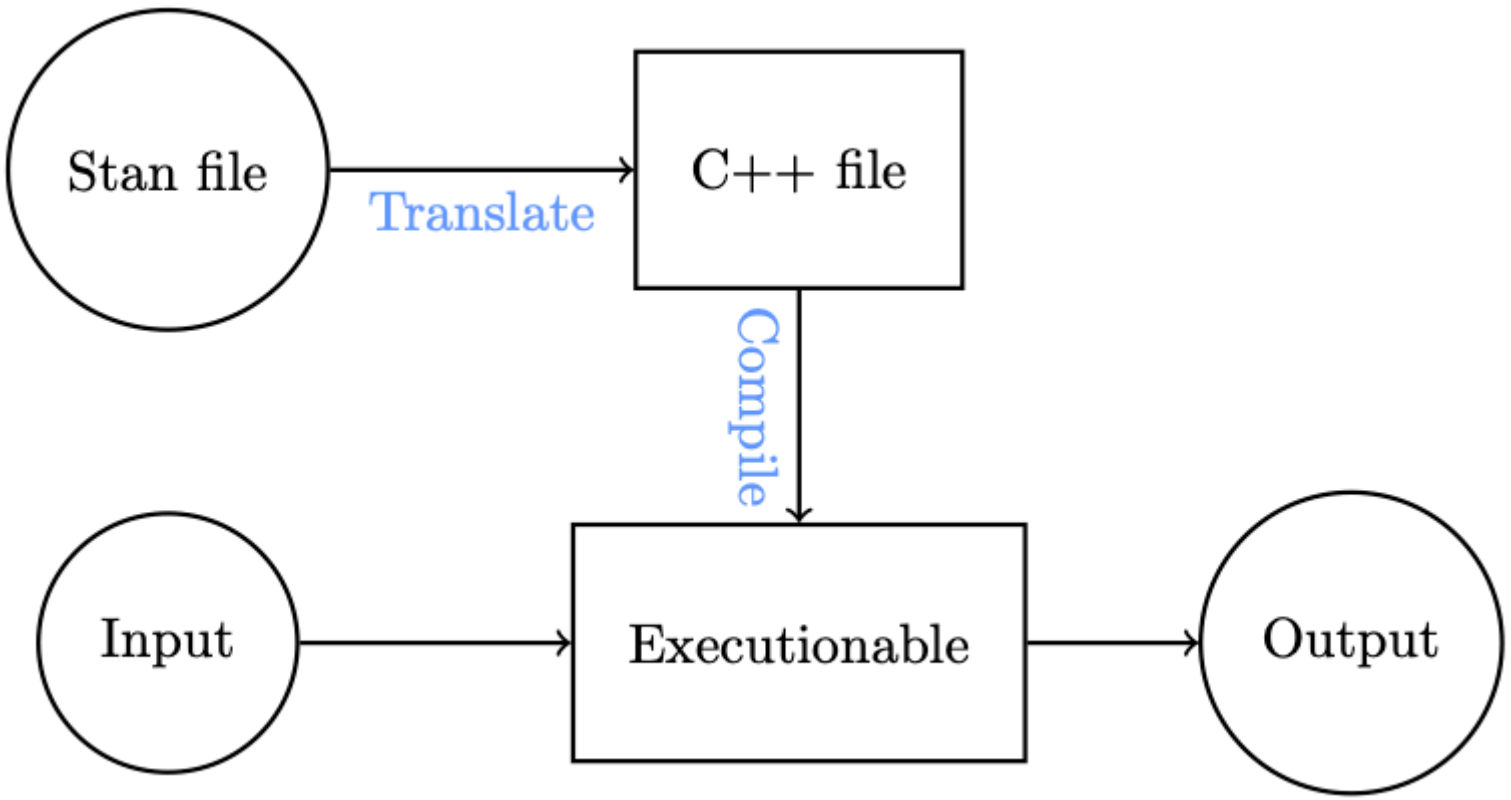
Stan in Python & R

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

- `pystan` & `RStan` - native language interface to the underlying Stan C++ libraries
  - 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 transforms of the data
}
parameters {
   // declares the model's parameters
}
transformed parameters {
   // allows variables to be defined in terms of data and parameters
}
model {
   // defines the log probability function
}
generated quantities {
   // allows derived quantities based on parameters, data, and random number generation
}
```

A basic example

Lec25/bernoulli.stan

```
data {
  int<lower=0> N;
  array[N] int<lower=0, upper=1> y;
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  theta ~ beta(1, 1); // uniform prior on interval 0,1
  y ~ bernoulli(theta);
}
```

Lec25/bernoulli.json

```
{
    "N" : 10,
    "y" : [0,1,0,0,0,0,0,0,0,1]
}
```

Build & fit the model

```
1 from cmdstanpy import CmdStanModel
2 model = CmdStanModel(stan_file='Lec25/bernoulli.stan')
```

```
1 fit = model.sample(data='Lec25/bernoulli.json', show_progress=False)
```

```
1 type(fit)
```

```
cmdstanpy.stanfit.mcmc.CmdStanMCMC
```

```
1 fit
```

```
CmdStanMCMC: model=bernoulli chains=4['method=sample', 'algorithm=hmc', 'adapt', 'engaged=1']
 csv_files:
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
 output_files:
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
    /var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpv2zs_85x/bernoulli6pe85w52/bernoulli-20250
```

Posterior samples

```
1 fit.stan_variables()
```

```
{'theta': array([0.45722, 0.36427, 0.36427, 0.52572, 0.31933, 0.2019 , 0.38248, 0.1655
       0.18252, 0.13986, 0.15708, 0.12406, 0.31417, 0.37685, 0.1857 , 0.27825, 0.2387
       0.20348, 0.17883, 0.17883, 0.12666, 0.28372, 0.09175, 0.1248 , 0.06877, 0.1458
```

```
1 np.mean( fit.stan_variables()["theta"] )
```

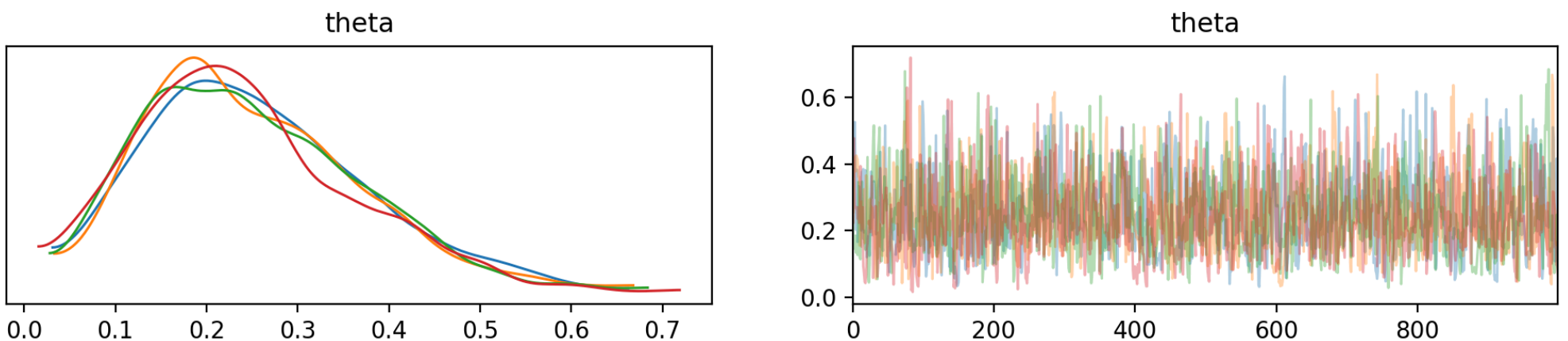
```
np.float64(0.24947370805)
```

Summary & trace plots

```
1  fit.summary()
```

	Mean	MCSE	StdDev	MAD	5%	50%	95%	ESS_bulk	ESS_tail
lp__	-7.241930	0.018553	0.685180	0.312658	-8.585620	-6.976330	-6.750170	1641.76	1882.85
theta	0.249474	0.003025	0.117231	0.117024	0.083825	0.232464	0.463878	1378.24	1718.05

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



Diagnostics

```
1 fit.divergences
```

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

```
1 fit.max_treedepths
```

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

```
1 fit.method_variables().keys()
```

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

```
1 print(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.
```

Gaussian process Example

GP model

Lec25/gp.stan

```
data {
  int<lower=1> N;
  array[N] real x;
  vector[N] y;
}
transformed data {
  array[N] real xn = to_array_1d(x);
  vector[N] zeros = rep_vector(0, N);
}
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);
  matrix[N, N] L = cholesky_decompose(add_diag(K, nug^2));
  // priors
  l ~ gamma(2, 1);
  s ~ cauchy(0, 5);
  nug ~ cauchy(0, 1);
  // model
```

Fit

```
d = pd.read_csv("data/gp2.csv").to_dict('list')
d["N"] = len(d["x"])
```

```
gp = CmdStanModel(stan_file='Lec25/gp.stan')
gp_fit = gp.sample(data=d, show_progress=False)
```

```
12:19:41 - cmdstanpy - INFO - CmdStan start processing
12:19:41 - cmdstanpy - INFO - Chain [1] start processing
12:19:41 - cmdstanpy - INFO - Chain [2] start processing
12:19:41 - cmdstanpy - INFO - Chain [3] start processing
12:19:41 - cmdstanpy - INFO - Chain [4] start processing
12:19:43 - cmdstanpy - INFO - Chain [2] done processing
12:19:43 - cmdstanpy - INFO - Chain [1] done processing
12:19:43 - cmdstanpy - INFO - Chain [3] done processing
12:19:43 - cmdstanpy - INFO - Chain [4] done processing
12:19:43 - cmdstanpy - WARNING - Non-fatal error during sampling:
Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp.stan', line 18, column 2
    Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp.stan', line 18, colum
Exception: gp_exp_quad_cov: length_scale is 0, but must be positive! (in 'gp.stan', line 17, colum
    Exception: gp_exp_quad_cov: length_scale is 0, but must be positive! (in 'gp.stan', line 17, c
    Exception: gp_exp_quad_cov: length_scale is 0, but must be positive! (in 'gp.stan', line 17, c
Consider re-running with show_console=True if the above output is unclear!
```

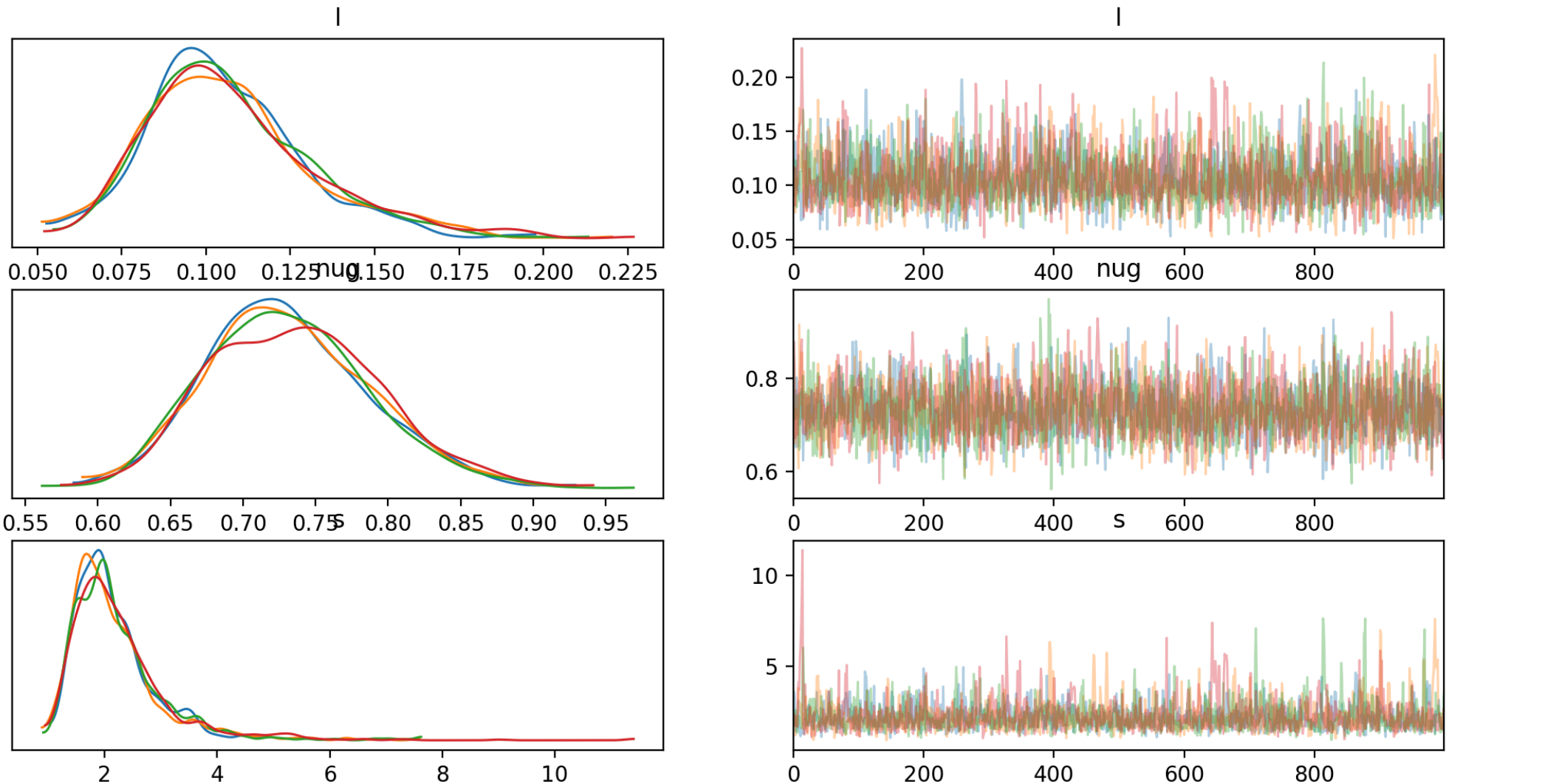
Summary

```
1 gp_fit.summary()
```

	Mean	MCSE	StdDev	MAD	5%	50%	95%	ESS_bulk	ESS_
lp__	-42.959300	0.030618	1.228470	1.044420	-45.378100	-42.665800	-41.588600	1620.19	2108
l	0.106533	0.000580	0.024411	0.021621	0.072973	0.102830	0.153564	2002.96	1784
s	2.199350	0.022507	0.823502	0.594738	1.321590	2.007950	3.681860	1845.11	1422
nug	0.732305	0.001148	0.057479	0.057420	0.645550	0.728543	0.832352	2597.64	2582

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.

```
1 import nutpie
2 m = nutpie.compile_stan_model(filename="Lec25/gp.stan")
3 m = m.with_data(x=d["x"],y=d["y"],N=len(d["x"]))
4 gp_fit_nutpie = nutpie.sample(m, chains=4)
```

Sampler Progress

Total Chains: 4

Active Chains: 0

Finished Chains: 4

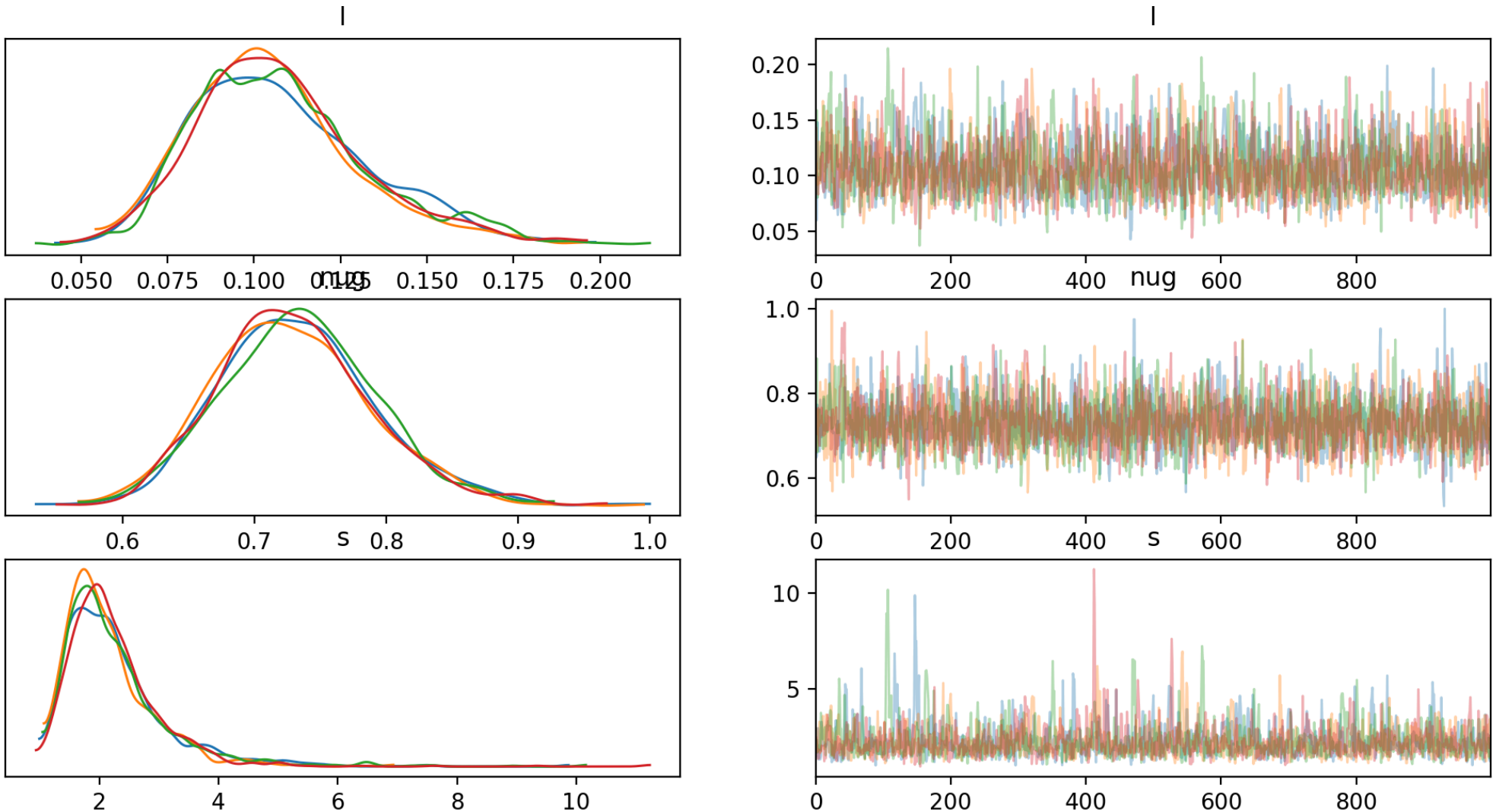
Sampling for now

Estimated Time to Completion: now

Progress	Draws	Divergences	Step Size	Gradients/Draw
	1400	0	0.82	3
	1400	0	0.80	3
	1400	0	0.80	1
	1400	0	0.79	3

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

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
l	0.107	0.025	0.067	0.160	0.001	0.000	1808.0	1840.0	1.0
nug	0.732	0.059	0.625	0.842	0.001	0.001	2751.0	2381.0	1.0
s	2.235	0.856	1.110	3.703	0.024	0.045	1664.0	1443.0	1.0


l
0.050
0.075
0.100
0.125
0.150
0.175
0.200
nug
0.6
0.7
s
0.8
0.9
1.0
2
4
6
8
10
l
0.20
0.15
0.10
0.05
0
200
400
nug
600
800
1.0
0.8
0.6
0
200
400
s
600
800
10
5
0
200
400
600
800

Performance

```
%%timeit -r 3
gp_fit = gp.sample(data=d, show_progress=False)
```

```
2.74 s ± 75.6 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

```
%%timeit -r 3
gp_fit_nutpie = nutpie.sample(m, chains=4, progress_bar=False)
```

```
1.21 s ± 12.3 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

Posterior predictive model

Lec25/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);
      for (n in 1:N1) {
        K[n, n] = K[n, n] + square(sigma);
      }
```

Posterior predictive fit

```python
d2 = pd.read_csv("data/gp2.csv").to_dict('list')
d2["N"] = len(d2["x"])
d2["xp"] = np.linspace(0, 1.2, 121)
d2["Np"] = 121
```

```python
gp2 = CmdStanModel(stan_file='Lec25/gp2.stan')
gp2_fit = gp2.sample(data=d2, show_progress=False)
```

```
12:20:09 - cmdstanpy - INFO - CmdStan start processing
12:20:09 - cmdstanpy - INFO - Chain [1] start processing
12:20:09 - cmdstanpy - INFO - Chain [2] start processing
12:20:09 - cmdstanpy - INFO - Chain [3] start processing
12:20:09 - cmdstanpy - INFO - Chain [4] start processing
12:20:12 - cmdstanpy - INFO - Chain [3] done processing
12:20:12 - cmdstanpy - INFO - Chain [2] done processing
12:20:12 - cmdstanpy - INFO - Chain [1] done processing
12:20:12 - cmdstanpy - INFO - Chain [4] done processing
12:20:12 - cmdstanpy - WARNING - Non-fatal error during sampling:
Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp2.stan', line 58, column 2
    Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp2.stan', line 58, colu
    Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp2.stan', line 58, colu
Exception: gp_exp_quad_cov: length_scale is 0, but must be positive! (in 'gp2.stan', line 57, colu
Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp2.stan', line 58, column 2
    Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp2.stan', line 58, colu
    Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp2.stan', line 58, colu
    Exception: cholesky_decompose: Matrix m is not positive definite (in 'gp2.stan', line 58, colu
Consider re-running with show_console=True if the above output is unclear!
```

Summary

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

	Mean	MCSE	StdDev	MAD	5%	50%	95%	ESS_bulk	ESS
lp__	-43.042800	0.032191	1.273380	1.072960	-45.623200	-42.734000	-41.602800	1615.63	230
l	0.106153	0.000558	0.024983	0.022396	0.070874	0.103053	0.153686	2101.05	202
s	2.192220	0.019064	0.819682	0.589771	1.294680	1.997890	3.735240	2269.50	188
nug	0.730838	0.001140	0.057326	0.058693	0.643901	0.727256	0.827427	2575.18	241
f[1]	3.469980	0.007242	0.440289	0.441466	2.722780	3.472880	4.183230	3726.82	383
...	...	...	...	...	...	...	...	...	...
f[117]	-0.607229	0.034081	2.037360	1.833990	-4.050970	-0.529064	2.534820	3628.10	373
f[118]	-0.564853	0.034749	2.086650	1.873460	-4.042150	-0.498740	2.639500	3649.49	381
f[119]	-0.520251	0.035192	2.128040	1.901170	-4.098390	-0.455036	2.803440	3693.62	363
f[120]	-0.474758	0.035421	2.162410	1.935030	-4.140110	-0.403482	2.907520	3764.51	357
f[121]	-0.429516	0.035444	2.190850	1.916190	-4.094190	-0.381014	3.005500	3846.13	358

125 rows × 10 columns

Draws

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

```
(4000, 121)
```

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

```
array([ 3.46998,  3.57005,  3.61907,  3.61183,  3.54474,  3.41625,  3.22692,  2.9796 ,  2.67934,
       -1.56055, -1.82155, -2.04347, -2.22579, -2.36835, -2.47109, -2.53395, -2.55682, -2.53968, -
       -0.29953, -0.01756,  0.25538,  0.5138 ,  0.75207,  0.96449,  1.14549,  1.28993,  1.39343,
        0.2243 ,  0.0885 , -0.01538, -0.08321, -0.11358, -0.10792, -0.07035, -0.00732,  0.07309,
        0.5379 ,  0.56842,  0.61659,  0.68387,  0.76956,  0.87107,  0.9842 ,  1.10373,  1.22392,
        1.32642,  1.17686,  1.00774,  0.82447,  0.63302,  0.43955,  0.24995,  0.06956, -0.09716, -
       -0.67891, -0.64584, -0.60723, -0.56485, -0.52025, -0.47476, -0.42952])
```

Plot