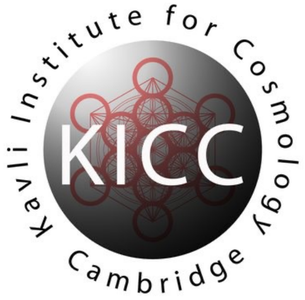
Diffusion meets Nested Sampling

Hills Coffee Talks

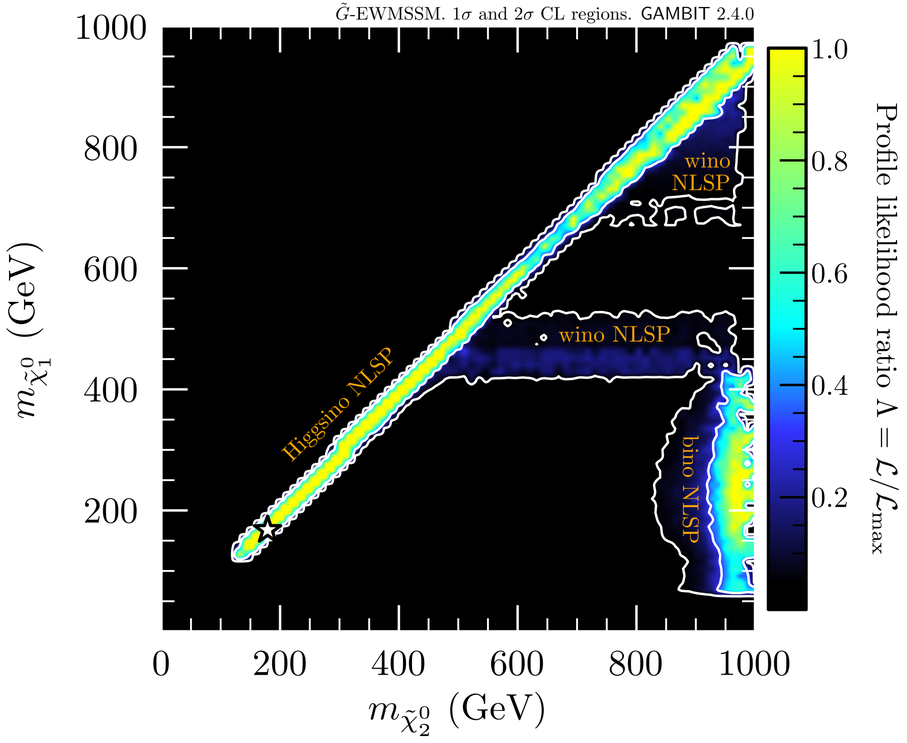


UNIVERSITY OF CAMBRIDGE

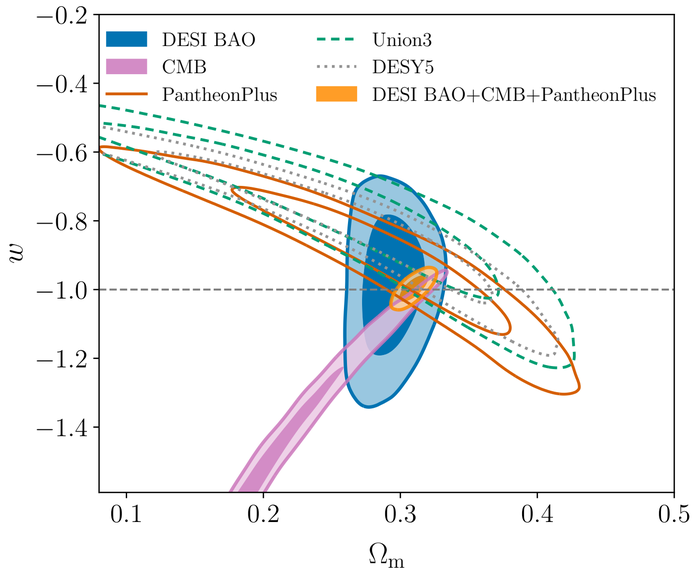
David Yallup - 21/05

INFERENCE

Fundamental physics is full of hard inference problems. Our optimization or sampling algorithms have to be able to navigate complex geometry



Gambit Collaboration SUSY profile likelihood scan[1]

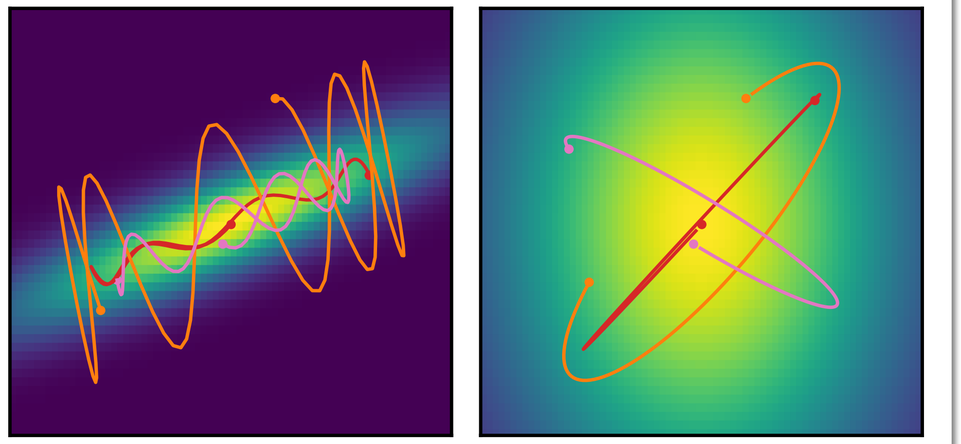


DESI collaboration LCDM samples[2]

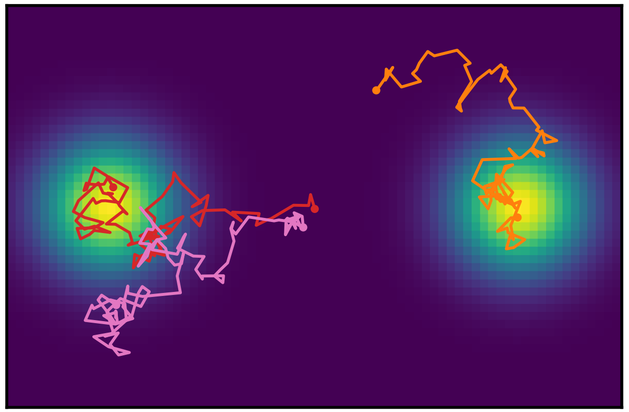
GEOMETRY

Bad geometry[3] in inference problems comes in many guises, and intuition gets progressively less clear in high dimension. Machine learnt neural mappings offer us a new tool to approach this.

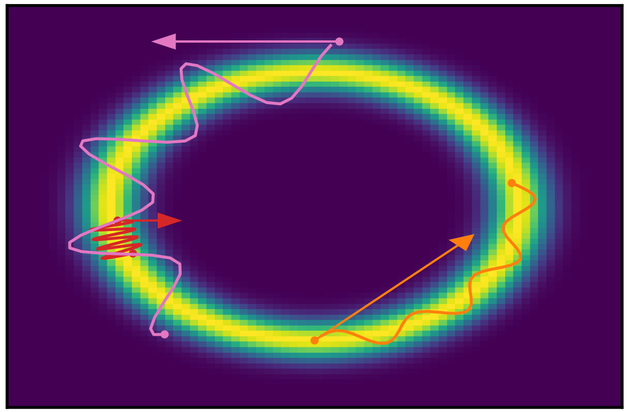
In the context of bridging distributions see `pocoMC`[4], `nessai`[5]



Whitening transforms to regularize the metric.



Clustering/ensembling to deal with multimodalities.



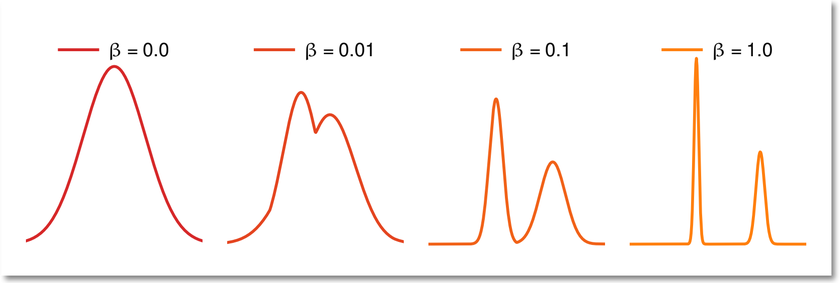
Gradients efficiently explore sweeping degeneracies.

BRIDGING DISTRIBUTIONS

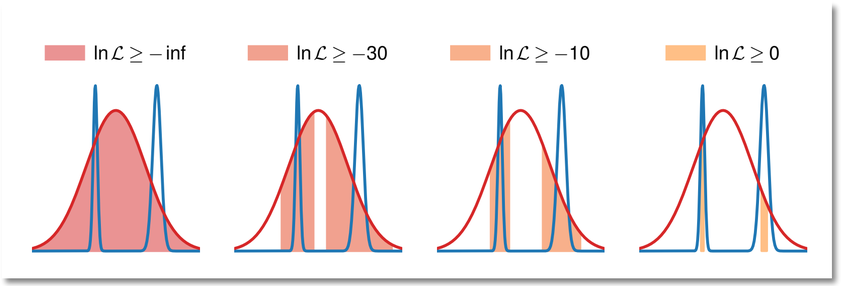
Population Monte Carlo methods — particle filters — form bridges from known (prior) to complex unknown (posterior) distributions. Sequential Monte Carlo (SMC) and Nested Sampling (NS) are two variants evolving populations of points[6]. Both give us access to the normalizing constant Z.

$$P(\theta \mid D) = \frac{\mathscr{L}(D \mid \theta) P(\theta)}{Z}$$

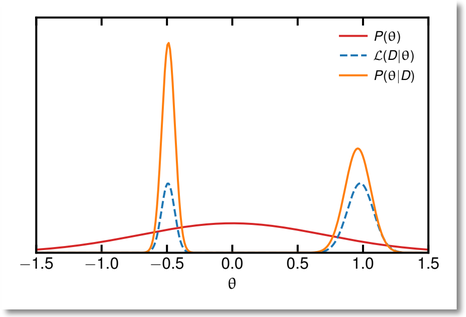
Bayes Rule



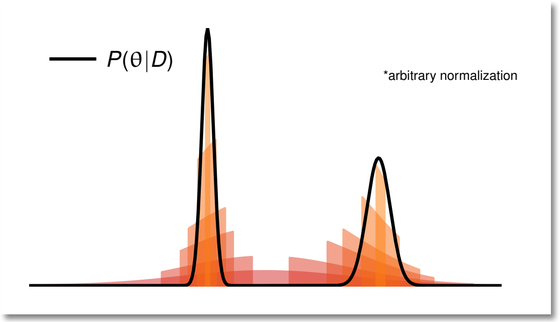
SMC



NS



Toy 1D inference problem.

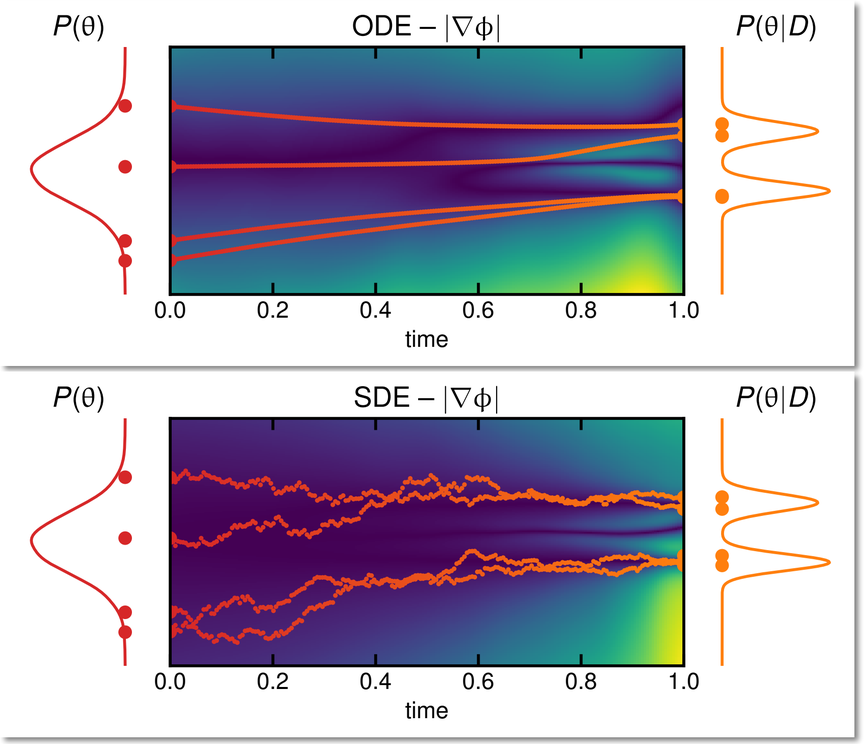


Schematic reconstructing posterior from NS shells.

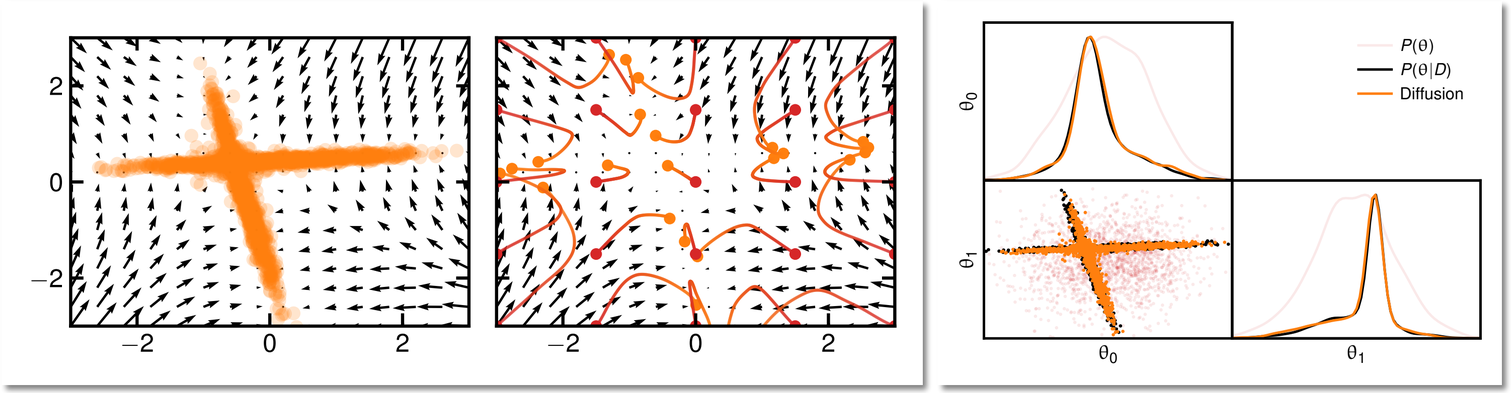
DIFFUSION

Diffusion models learn the gradient of the implicit density of a point cloud. Solving evolution through this field with Stochastic Differential Equation (SDE) or Ordinary Differential Equation (ODE) solvers yields Diffusion[7] or Continuous flows[8].

Neural learnt maps can transport any known distribution to an implicit target, no strict requirement on latent/prior!



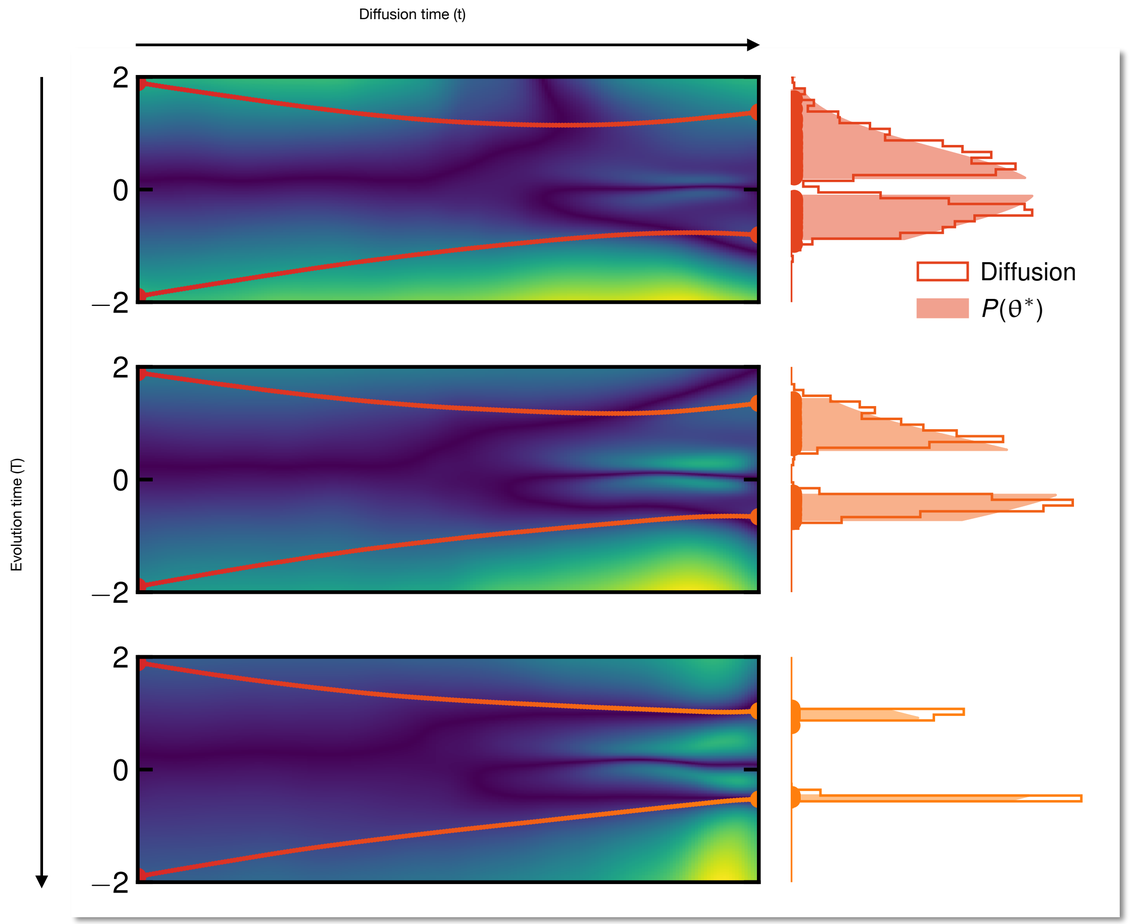
Learnt gradient vector field mapping prior to posterior.



2D representation of learnt vector fields.

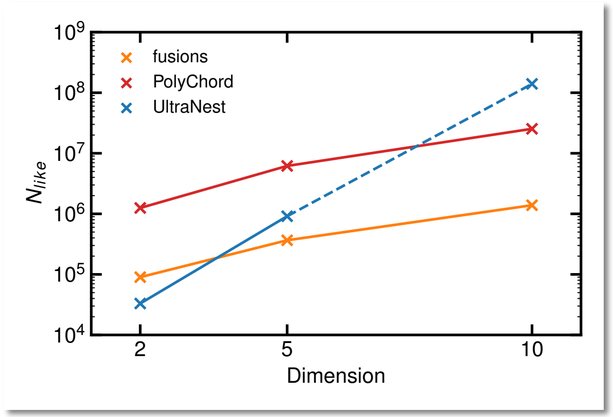
RESULTS*

Diffusion models introduce time axis to the problem, bridging algorithms have another time axis we can efficiently evolve by fine tuning the score estimate.

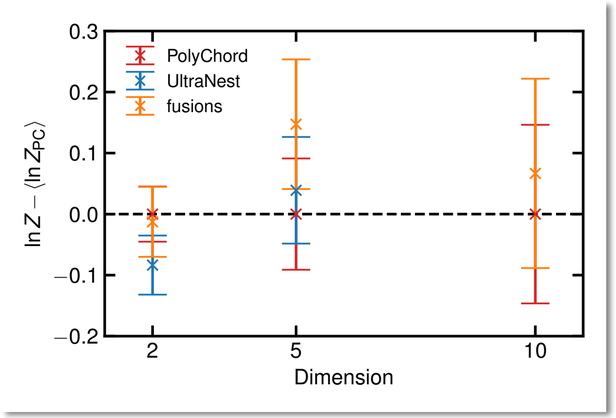


Evolution of likelihood constrained prior through a NS run.

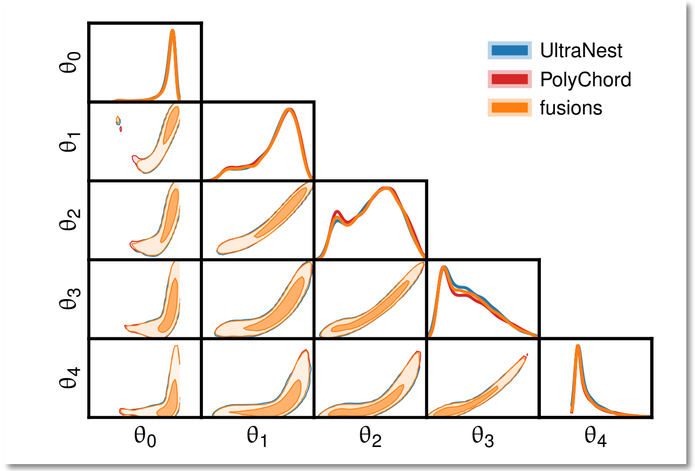
RESULTS*



Number of function evaluations required for Rosenbrock function in various dimensions



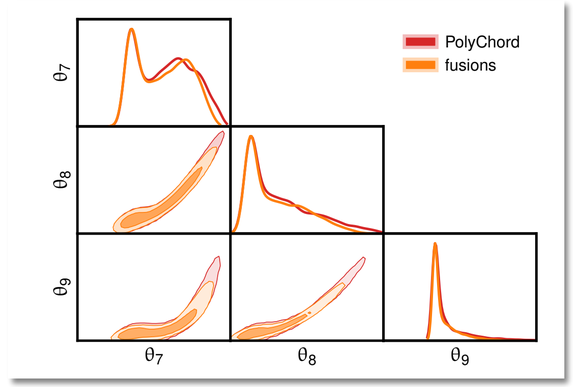
Calculated log integral for Rosenbrock function in various dimensions



Corner plot of 5D Rosenbrock function

$N_{\text{live}} = 2000$ for all, otherwise following defaults.

UltraNest in 10D N_{like} projected after early termination due to exceeding walltime.



Partial corner plot of 10D Rosenbrock function

Comparison to standard (non-neural) tools[9,10] shows promising scaling, comparable to step samplers despite using rejection sampling, whilst maintaining accurate predictions on benchmark challenging problems.

Algorithm demonstrated uses *zero* classical methods, treating the geometry of the problem solely with neural networks and score based models.

* Work in progress, comparison to other neural methods[4,5,11,12], plenty left on the table to tune in the algorithm.

DIFFUSION MEETS NESTED SAMPLING

NEUTRALISING BAD GEOMETRY IN BRIDGING INFERENCE PROBLEMS

DAVID YALLUP

yallup/fusions

dy297@cam.ac.uk

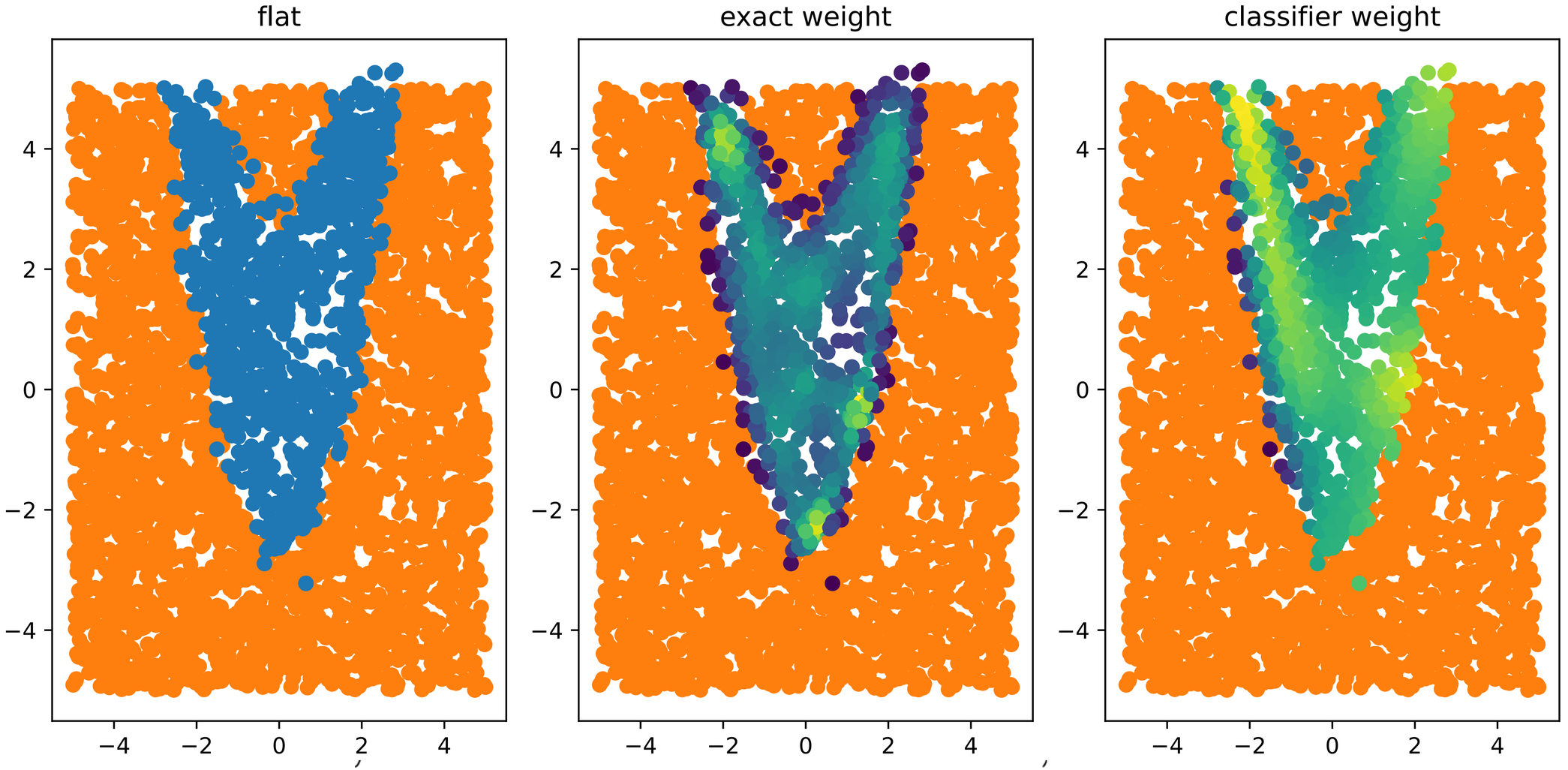
yallup@github.io

References:

1.[2303.09082] The Gambit collaboration
2.[2404.03002] DESI Collaboration
3.[1903.03704] Hoffman et al.
4.[2207.05660] Karamanis et al.
5.[2102.11056] Williams et al.
6.[2205.15570] Ashton et al.
7.[2011.13456] Song et al.
8.[2302.00482] Tong et al.
9.[2101.09604] Buchner
10.[1506.00171] Handley et al.
11.[2306.16923] Lange
12.[1903.10860] Moss

Technical references:

- github.com/patrick-kidger/diffrax
- github.com/handley-lab/anesthetic
- github.com/yallup/fusions
- github.com/google/flax
- github.com/google/jax

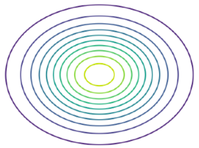


$$\nabla \log p_t(x_t|y) = \underbrace{\nabla \log p_t(x_t)}_{\text{pre-trained score}} + \underbrace{\nabla \log c_t(y|x_t)}_{\text{guidance}},$$

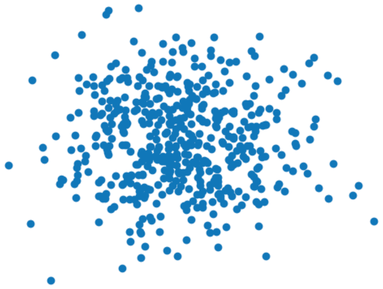

S'
π
S
l*

Backup

Modelling distributions



$$\frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$



Distribution

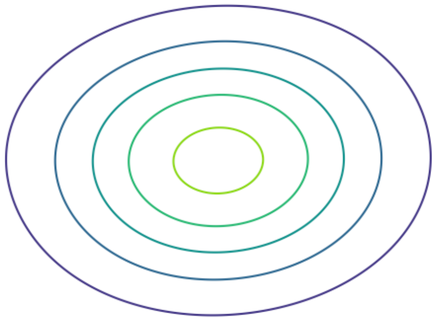
Samples

Can be represented by

.rvs
PolyChord.run
emcee.sample

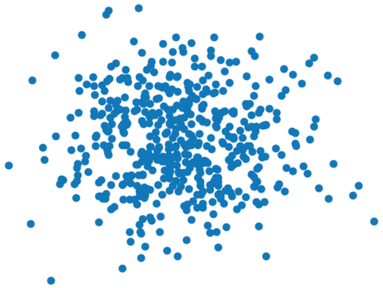
Implicitly defines

GP.fit
KDE.fit
NormalisingFlow



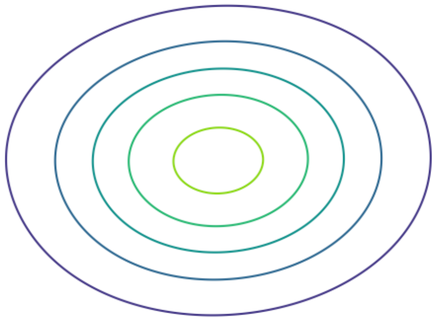
Modelling distributions

If You have the full loop you can “Emulate”.
If you only have Samples, the ability to bestow them with a distribution is useful!



Distribution

Samples

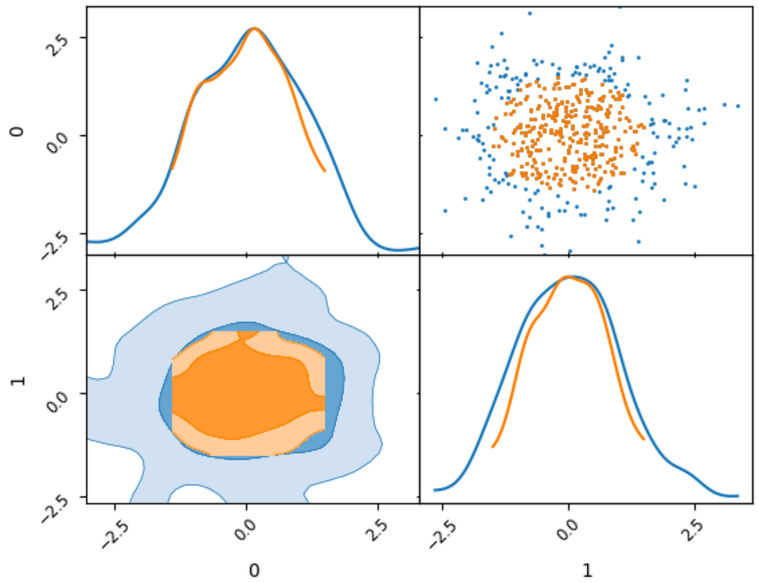


Implicitly defines

GP.fit
KDE.fit
NormalisingFlow

Nested Sampling with distributions

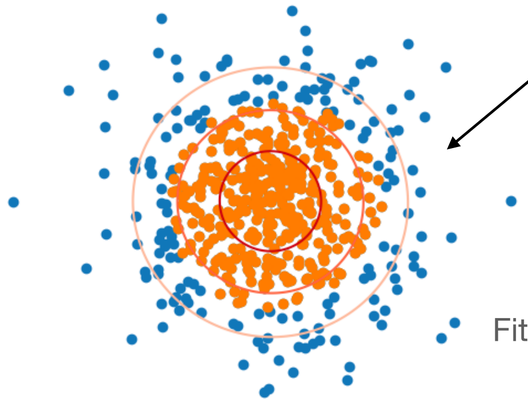
“Rejection sampling”



- Blue Samples from a known Distribution
- Orange Samples from likelihood constrained region

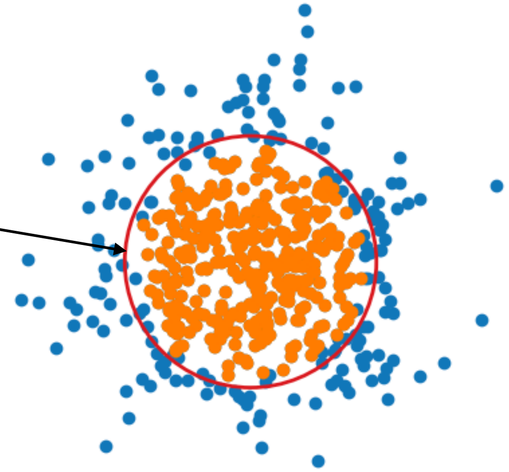
Orange implicitly defines a distribution we want more of

How can we “bestow” these points with a distribution?



Fit a neural density: nessai etc.

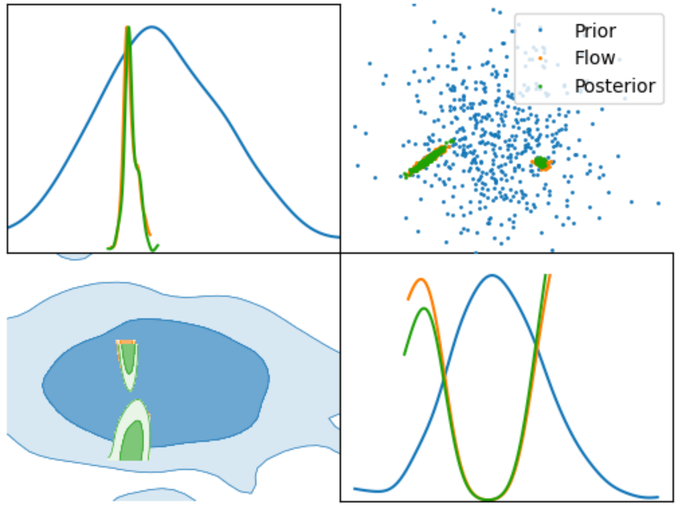
Fit an ellipse: MultiNest

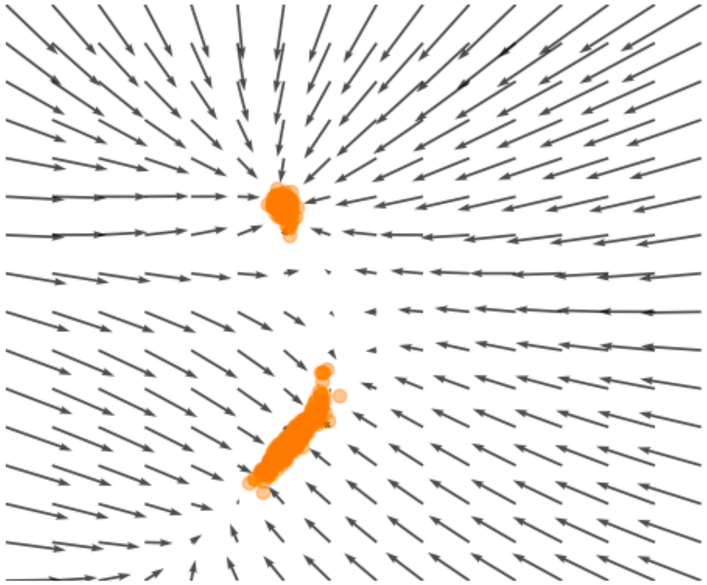


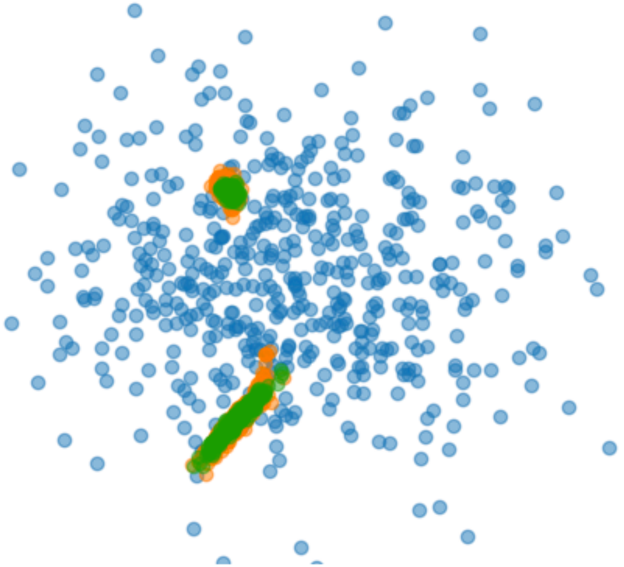
Diffusion Models

The current vogue in density estimation

Optimal Transport, Diffusive processes, Continuous flows, Score based generative models, flow matching etc.etc. All the same thing!



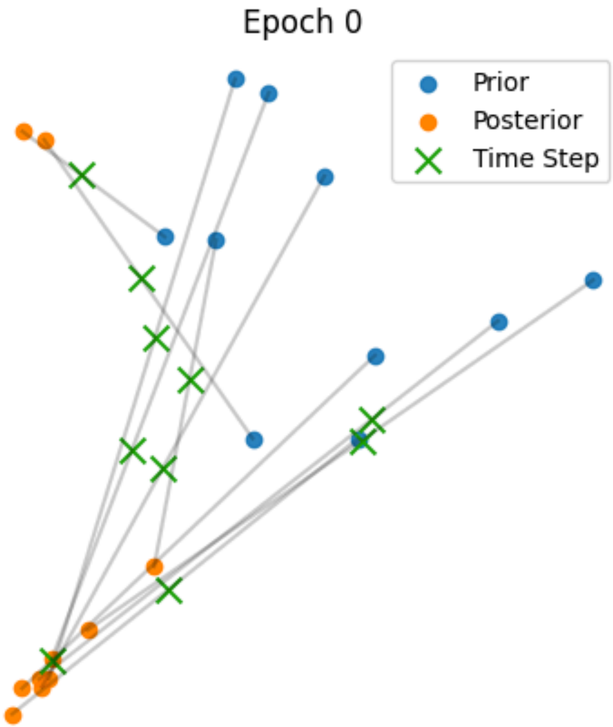


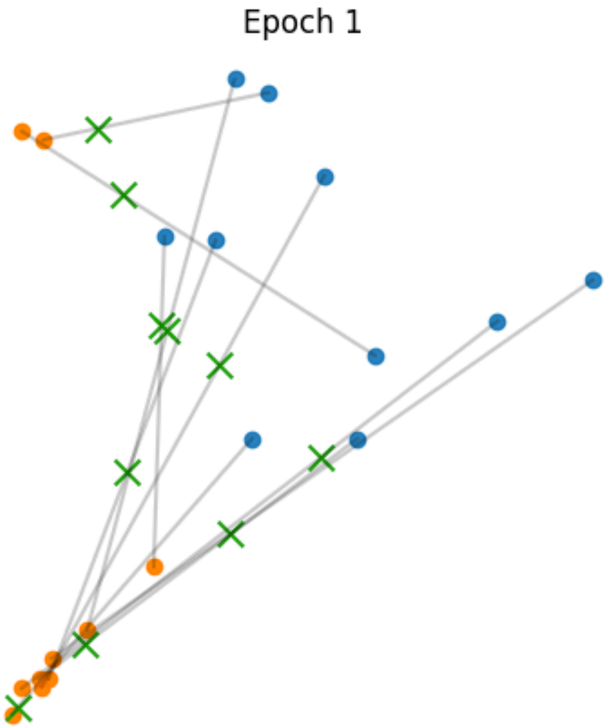


Neural Networks Learn gradient fields (Score) that map one distribution to another

How it works

[2302.00482] Flow matching, incredibly simple objective!



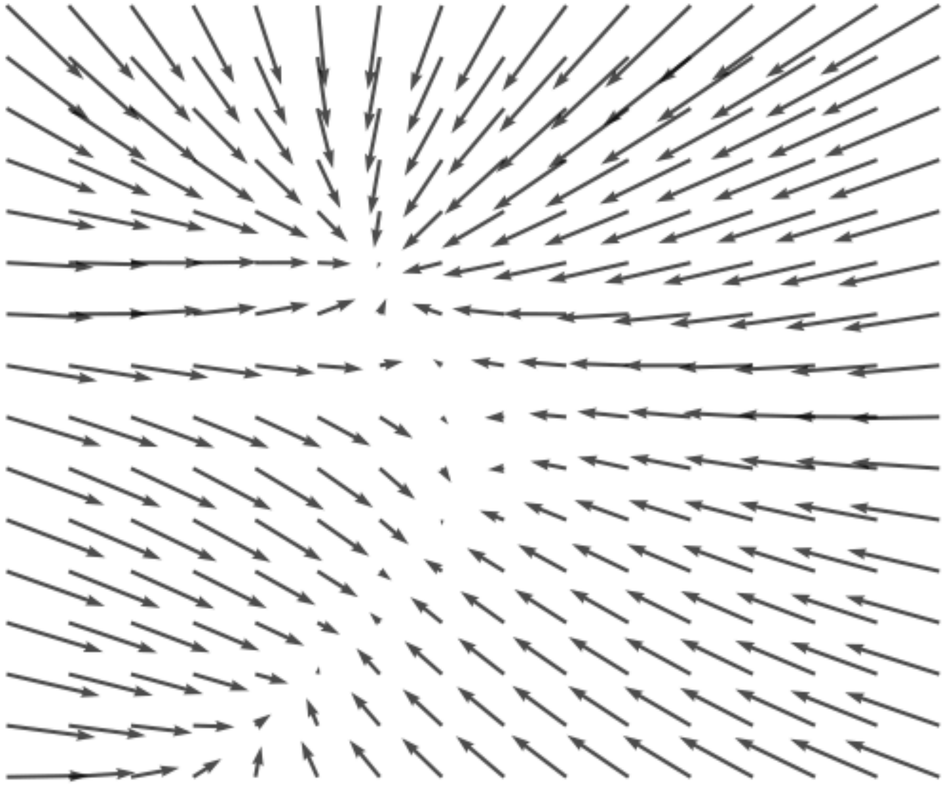



```python
@partial(jit, static_argnums=[0])
def loss(self, params, batch, batch_prior, batch_stats, rng):
    """Loss function for training the CFM score.

    Args:
        params (jnp.ndarray): Parameters of the model.
        batch (jnp.ndarray): Target batch.
        batch_prior (jnp.ndarray): Prior batch.
        batch_stats (Any): Batch statistics (batchnorm running totals).
        rng: Jax Random number generator key.

    """
    # sigma_noise = 1e-3
    rng, step_rng = random.split(rng)
    N_batch = batch.shape[0]

    t = random.uniform(step_rng, (N_batch, 1))
    noise = random.normal(step_rng, (N_batch, self.ndims))
    psi_0 = t * batch + (1 - t) * batch_prior + self.noise * noise

    output, updates = self.state.apply_fn(
        {"params": params, "batch_stats": batch_stats},
        psi_0,
        t,
        train=True,
        mutable=["batch_stats"],
    )
    psi = batch_prior - batch
    loss = jnp.mean((output - psi) ** 2)
    return loss, updates
```

- Pair up a random set of prior and target samples
- Generate a random time step for each pair, $t \leftarrow [0,1]$
- Simple MSE regression on the prior-target vectors and a neural network learning the vector field at the time step

By shuffling and simulating a new time step we build up coverage of the whole space and trace out all paths

What to do with a learned vector field



Diffusion:

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t \sqrt{1-\beta_t} x_{t-1}, \beta_t I)$$

$$q(x_{1:T} \mid x_0) = \prod_{t=1}^{T} q(x_t \mid x_{t-1})$$

Solve a SDE, easy to simulate solutions from
[yang-song.net/blog/2021/score]

Continuous Flows:

$$\frac{dy}{dt} = \nabla_\theta f(t, y(t)), \quad y(0) = y_0$$

Solve an ODE, slightly harder but nicer properties for science
[2202.02435]

Jacobians are an interesting part of this

Often for scientific applications need this