From Molecules to Maps: A Hands-On Tour of Dimensionality Reduction techniques for Molecular Dynamics

A quick journey through how CVs have grown from simple physics rules to sophisticated data-driven representations—unlocking deeper insight into molecular dynamics.

~ Developed by Shaheerah Shahid (JRF, SNBNCBS)

working with Prof. Suman Chakrabarty

Email: shaheerah.shahid@bose.res.in

```
!python3 --version
```

```
Python 3.12.12
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
import os
os.chdir('/content/drive/MyDrive/smcb_data')
```

```
# ## Install required python modules
# !pip install torch --index-url https://download.pytorch.org/whl/cuda
!pip -q install MDAnalysis
!pip -q install mdtraj
!pip -q install nglview
!pip -q install deeptime
!pip -q install SciencePlots
```

```
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependen
datasets 4.0.0 requires fsspec[http]<=2025.3.0,>=2023.1.0, but you have fsspec 2025.9.0 which is incompatible.
numba 0.60.0 requires numpy<2.1,>=1.22, but you have numpy 2.2.6 which is incompatible.
tensorflow 2.19.0 requires numpy<2.2.0,>=1.26.0, but you have numpy 2.2.6 which is incompatible.
```

```
import MDAnalysis as mda
import numpy as np
import mdtraj as md
import nglview as nv
import deeptime
import matplotlib.pyplot as plt
from google.colab import output
output.enable_custom_widget_manager()
```

```
WARNING:MDAnalysis.coordinates.AMBER:netCDF4 is not available. Writing AMBER ncdf files will be slow.
```

```
# Plotting tools and params
import scienceplots
from plot2d import plot_free_energy
plt.style.use(['science', 'ieee', 'grid'])
plt.rcParams["text.usetex"] = False
plt.rcParams["font.family"] = "serif"
plt.rcParams["font.serif"] = ["DejaVu Serif"]  # fallback
plt.rcParams["figure.dpi"] = 120
plt.rcParams["figure.figsize"] = (6.5, 4.2)
```

```
structure = mda.Universe( "2JOF-0-protein.pdb" )
```

```
view = nv.show_mdanalysis(structure)
view
```

```
traj = mda.Universe("2JOF-0-protein.pdb","aligned_traj_skip10.xtc")
print(traj.trajectory)
```

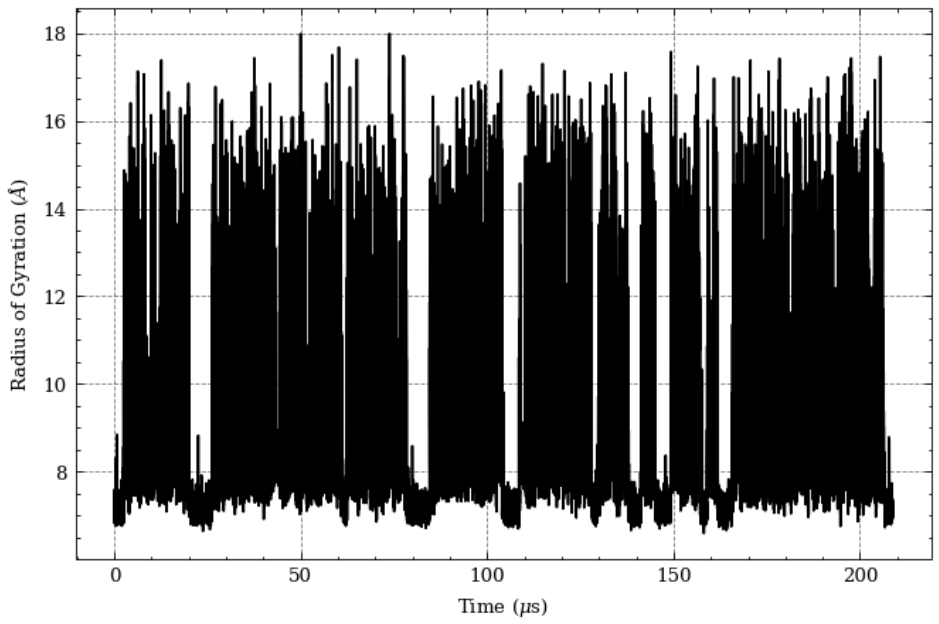
```
<XTCReader aligned_traj_skip10.xtc with 104400 frames of 272 atoms>
/usr/local/lib/python3.12/dist-packages/MDAnalysis/coordinates/XDR.py:261: UserWarning: Reload offsets from trajectory
 ctime or size or n_atoms did not match
  warnings.warn(
```

```
time = np.arange(len(traj.trajectory)) * 2 * 0.001# in ns
```

```
protein = traj.select_atoms('protein')
rog = np.zeros(104400, dtype=float)
for i, ts in enumerate(traj.trajectory):
  rog[i] = protein.radius_of_gyration()
```

```
plt.plot(time,rog)
plt.xlabel(r'Time ($\mu$s)')
plt.ylabel('Radius of Gyration ($\\AA$)')
```

```
Text(0, 0.5, 'Radius of Gyration ($\\AA$)')
```



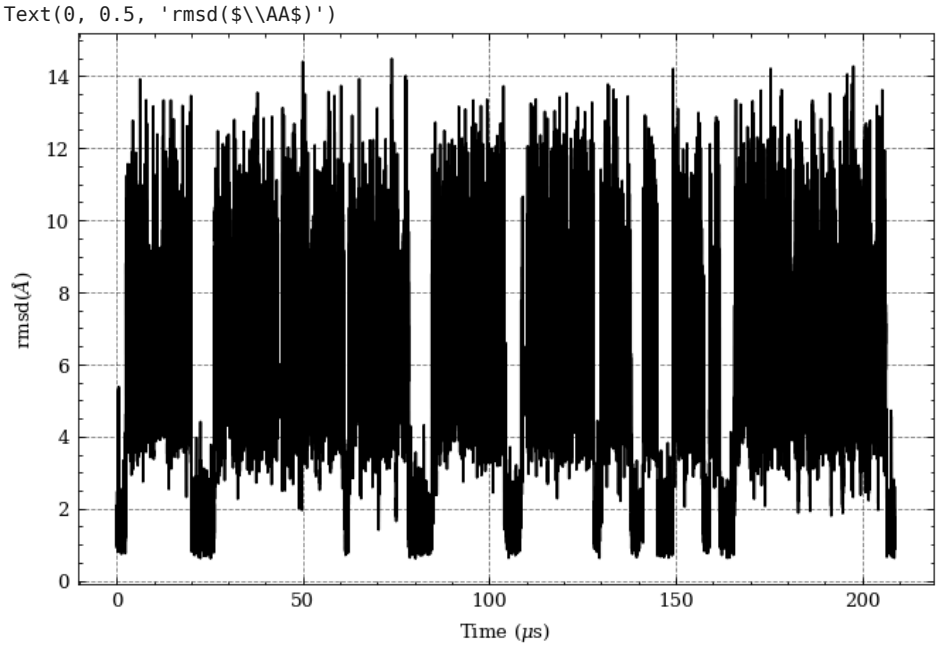

```
from MDAnalysis.analysis import rms
R = rms.RMSD(traj,  # trjectory to align
         structure,  # reference
         select='backbone') # selection
R.run()
rmsd = R.results.rmsd[:,2]
```

```
rmsd
```

```
array([0.92672413, 1.38318544, 1.3153444 , ..., 1.30054452, 1.17413269,
       1.34848922], shape=(104400,))
```

```
plt.plot(time,rmsd)
plt.xlabel(r'Time ($\mu$s)')
plt.ylabel('rmsd($\\AA$)')
```

```
Text(0, 0.5, 'rmsd($\\AA$)')
```




```
import os

os.path.exists("/content/drive/MyDrive/smcb_data/fnc.py")
```

```
True
```

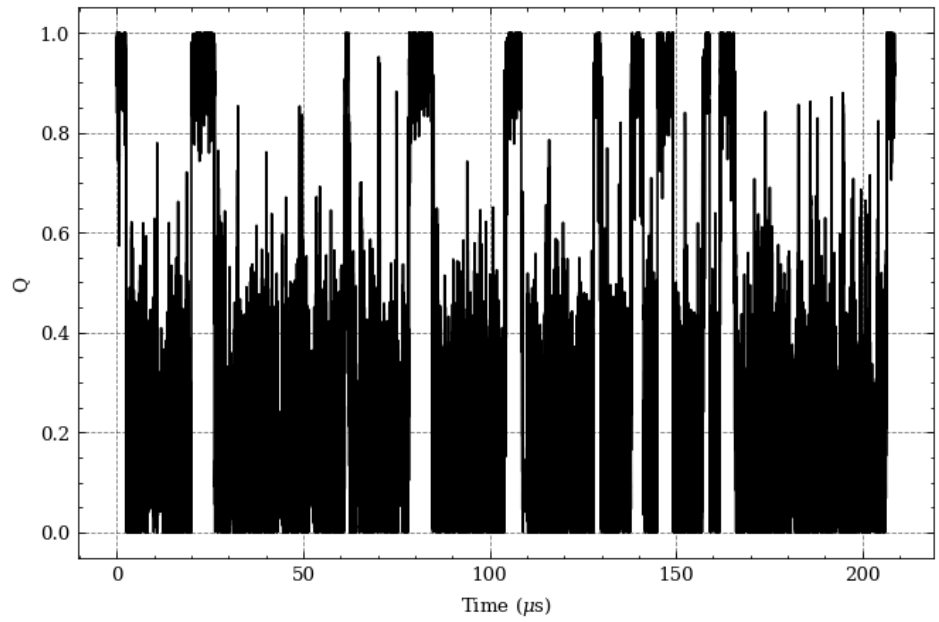
```
from fnc import best_hummer_q
```

```
q = best_hummer_q("aligned_traj_skip10.xtc", top_file="2JOF-0-protein.pdb", native_file= "2JOF-0-protein.pdb")[0]
```

```
/content/drive/MyDrive/smcb_data/fnc.py:94: RuntimeWarning: overflow encountered in exp
  q = np.mean(1.0 / (1 + np.exp(BETA_CONST * (r - LAMBDA_CONST * r0))), axis=1)
```

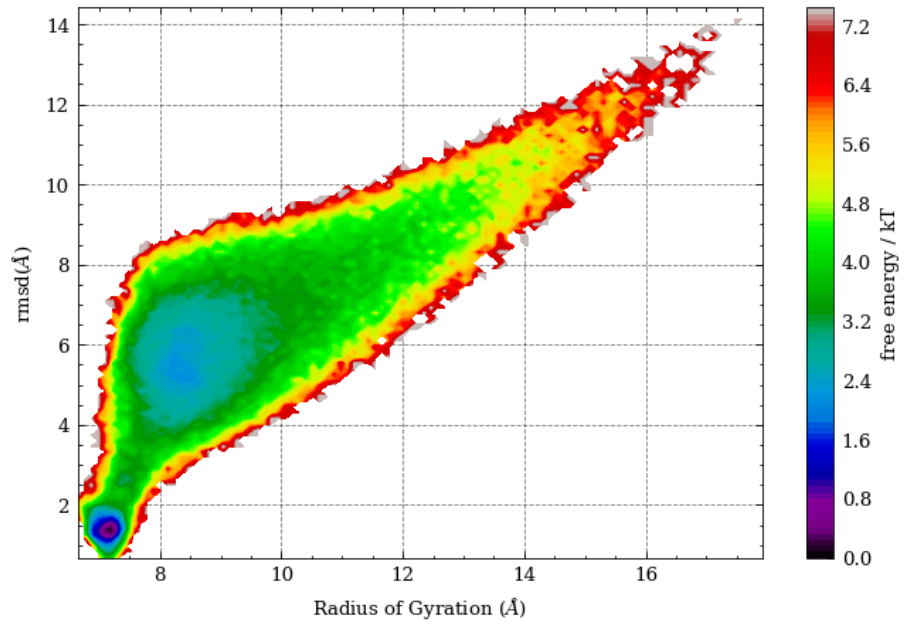
```
plt.plot(time,q)
plt.xlabel(r'Time ($\mu$s)')
plt.ylabel('Q')
```

```
Text(0, 0.5, 'Q')
```



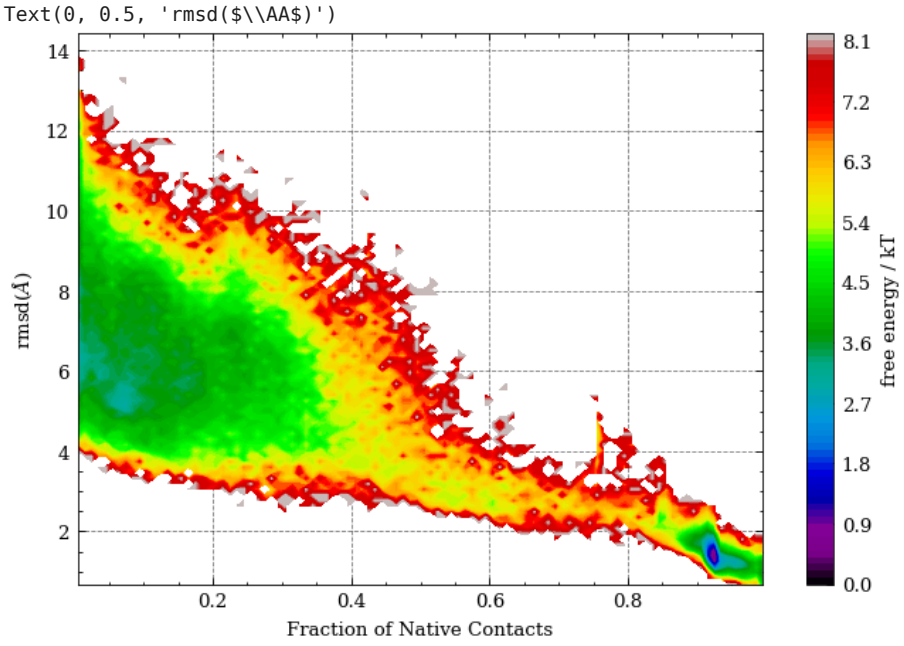

```
plot_free_energy(rog,rmsd)
plt.xlabel("Radius of Gyration ($\\AA$)")
plt.ylabel('rmsd($\\AA$)')
```

```
Text(0, 0.5, 'rmsd($\\AA$)')
```

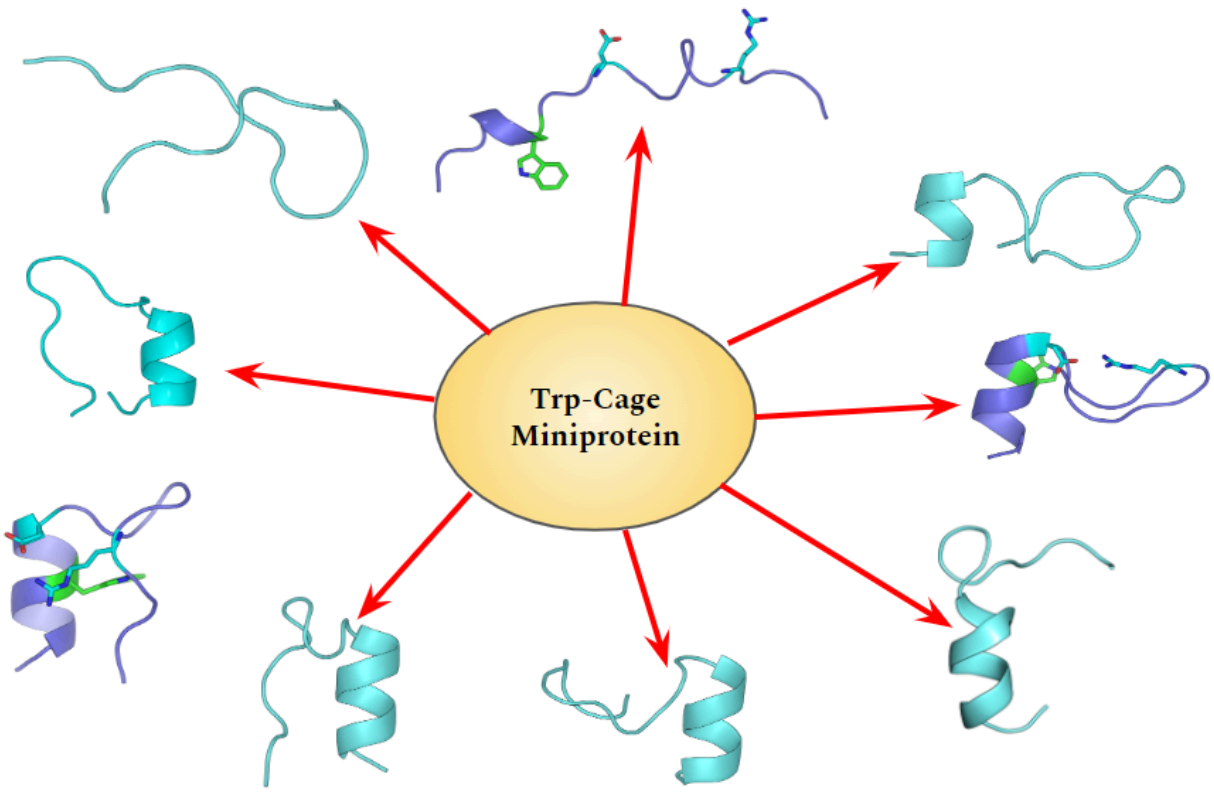



```
plot_free_energy(q,rmsd)
plt.xlabel("Fraction of Native Contacts")
plt.ylabel('rmsd($\\AA$)')
```

```
Text(0, 0.5, 'rmsd($\\AA$)')
```



But A protein has many conformers....

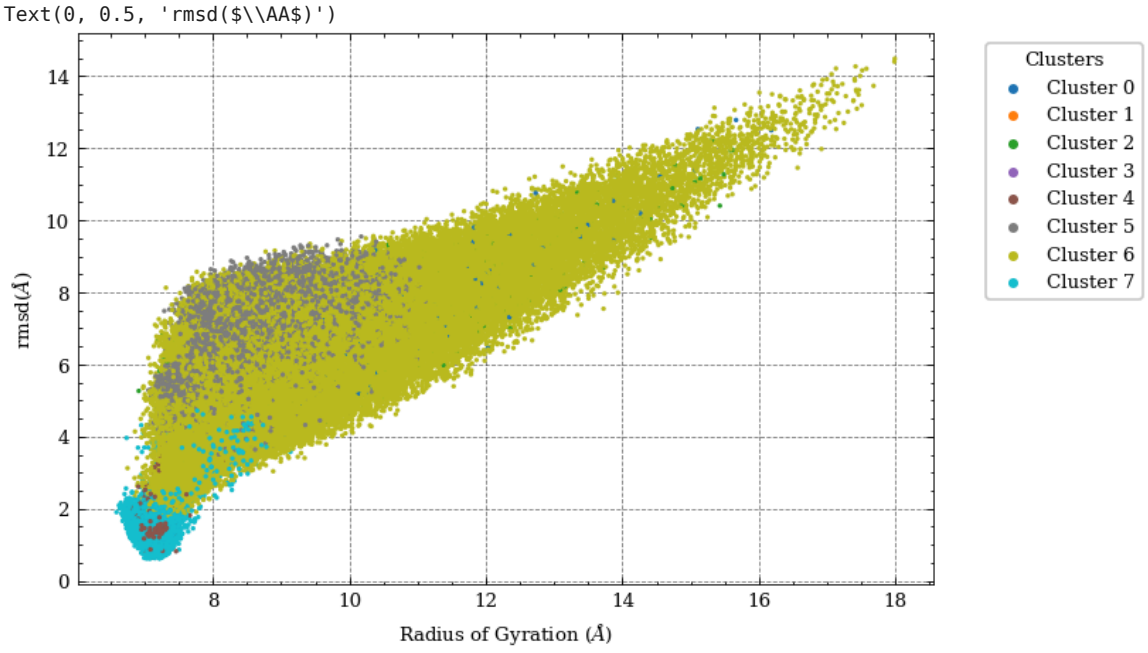



```
color = np.loadtxt("msm_8states_my.txt")
# Compute unique clusters and generate colors dynamically
unique_clusters = np.unique(color)
print(unique_clusters)

n_clusters = len(unique_clusters)

# Build a discrete colormap with exactly n_clusters colors
base_cmap = plt.get_cmap('tab10')
colors = base_cmap(np.linspace(0, 1, n_clusters))

# Map each cluster ID to its color
cluster_to_color = {cl: colors[i] for i, cl in enumerate(unique_clusters)}
color_values = np.array([cluster_to_color[c] for c in color[::10]])
```

```
[0. 1. 2. 3. 4. 5. 6. 7.]
```

```
# Create plot
fig, ax = plt.subplots()
ax.scatter(rog,rmsd,c=color_values, s=1)
# Custom legend instead of continuous colorbar
for cl, col in cluster_to_color.items():
    ax.scatter([], [], color=col, label=f'Cluster {int(cl)}', s=10)
ax.legend(title="Clusters", bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xlabel("Radius of Gyration ($\\AA$)")
plt.ylabel('rmsd($\\AA$)')
```

```
Text(0, 0.5, 'rmsd($\\AA$)')
```




```
for c in unique_clusters:
    idx = np.where(color[::10] == c)[0]
    mean_r = np.mean(rmsd[idx])  # rmsd column 2 = RMSD in Å
    print(f"Color {int(c)}: mean RMSD = {mean_r:.3f} Å")
```

```
Color 0: mean RMSD = 7.247 Å
Color 1: mean RMSD = 5.123 Å
Color 2: mean RMSD = 6.460 Å
Color 3: mean RMSD = 5.246 Å
Color 4: mean RMSD = 1.488 Å
Color 5: mean RMSD = 6.666 Å
Color 6: mean RMSD = 6.373 Å
Color 7: mean RMSD = 1.473 Å
```

The remainder of the tutorial introduces a simple, practical framework for thinking about **data-driven order parameters**, built around one key idea:

A projection = (1) what data you feed in + (2) how your model transforms it.

Data + Model ⟶ CVs

---

1. What Is Your Data? ⟶ Featurization

Before any model can discover order parameters, you must decide **what information about your system is worth presenting to it**.
This step is called **featurization**, and it determines the *space* you are projecting from.

Common featurizations include:

- **Geometric features:** distances, angles, dihedrals
- **Internal coordinates:** backbone torsions, side-chain χ angles
- **Contact maps:** binary or continuous
- **Environment descriptors:** solvent exposure, hydrogen-bond counts
- **Alignment-invariant features:** pairwise distances, inverse distances
- **High-level embeddings:** graph features, symmetry-preserving features

Featurization should:

- Preserve **relevant physical information**
- Remove **irrelevant degrees of freedom** (translations/rotations)
- Provide a **consistent representation** across frames
- Be computationally efficient for large trajectories

Think of it as **choosing the raw ingredients** for your data-driven CV.

```
from MDAnalysis.analysis.distances import distance_array, self_distance_array
# Ca-Ca distances as features
ag = structure.select_atoms('name CA')
self_distance_array(ag, structure.dimensions), self_distance_array(ag, structure.dimensions).shape
```

```
(array([ 3.86088571,  5.94451728,  8.18361839,  8.86015312, 10.13306004,
        12.02977775, 13.43374714, 14.57334692, 16.004756  , 14.37538085,
        16.51619148, 19.45426304, 19.39455185, 18.25497282, 15.10078331,
        13.32902745, 10.15903137,  7.36954493,  5.85440754,  3.92080566,
         5.66084295,  5.20458865,  6.55712152,  8.90206305,  9.98130976,
        10.78398969, 12.49202794, 11.13062726, 13.77328453, 16.31242969,
        16.0818122 , 15.43756278, 12.2633651 , 11.32689267,  8.57883631,
         6.27935344,  6.94242341,  3.86274242,  5.45417036,  5.36663411,
         6.34060355,  8.6627415 ,  9.99801793, 10.59324298,  9.21973214,
        11.13466252, 13.9768063 , 14.55912894, 13.7907886 , 11.35101783,
        10.20673388,  6.86923427,  6.37794719,  7.76621016,  3.85625701,
         5.58397861,  5.03283188,  5.95117533,  8.49848273,  9.66161594,
         9.77511986, 12.05778982, 14.10166366, 14.99795704, 15.20244109,
        13.1047141 , 12.94681225, 10.09453002,  9.89277538, 11.40742703,
         3.87607382,  5.47623249,  5.03056918,  6.12962586,  8.52494975,
         8.64060849, 11.85049967, 13.37985227, 13.35354602, 13.943495  ,
        11.36475505, 11.99132249, 10.03820676,  9.35784494, 11.48516105,
         3.88827438,  5.5328135 ,  5.05851109,  6.0113005 ,  4.94150245,
         8.15770231,  9.99998737,  9.94052267, 10.14553094,  7.77335034,
         8.56059861,  7.1068552 ,  7.6425197 , 10.74252539,  3.87337729,
         5.38379576,  4.91602031,  5.59991991,  7.67891255,  9.19455606,
        10.52702812, 11.23132491, 10.08917245, 10.8807331 ,  9.07346339,
        10.50342711, 13.28761506,  3.8308237 ,  5.53791936,  7.81656437,
        10.55954704, 10.93724892, 11.76602365, 13.44239075, 12.13336689,
        13.68190941, 12.27051576, 12.98549245, 15.65415845,  3.82396903,
         5.89388236,  9.24780489,  9.02915825,  8.79566426, 10.88409046,
         9.52153254, 11.92862872, 11.59798387, 12.44038643, 15.66018675,
         3.84412199,  6.06486702,  5.50072616,  6.48130568,  8.59481079,
         8.50871287, 10.88152593, 10.80720583, 12.78845547, 16.24900701,
         3.79277808,  5.28061438,  5.39700712,  5.73280537,  5.04217652,
         7.05245063,  7.38281342,  9.74474585, 13.41069187,  3.82362148,
         5.68329508,  4.88916219,  6.3766301 ,  7.47121476,  8.02709954,
        11.38234303, 14.86226229,  3.80621766,  5.53642683,  7.9721889 ,
        10.36201533, 11.51899945, 14.55044519, 18.19655338,  3.88849644,
         5.89251117,  9.27736022, 11.41104385, 13.89946416, 17.70667105,
         3.89901552,  6.26490828,  9.01624166, 11.86264734, 15.54083483,
         3.89544575,  6.71572015,  8.65548626, 12.40261364,  3.87184472,
         6.18222037,  9.57831058,  3.82538822,  6.98523328,  3.81771806]),
 (190,))
```

```
## Calculating distances/features for the trajectory..
distance_array = np.zeros((traj.trajectory.n_frames, 190))
ag = traj.select_atoms('name CA')
for i, ts in enumerate(traj.trajectory):
  distance_array[i] = self_distance_array(ag, traj.dimensions)
```

```
print(distance_array.shape)
data = distance_array
```

```
(104400, 190)
```

⚙️ 2. What Do You Want to Look At? ⟶ The Model

Once your trajectory is featurized, you choose **a model that defines the type of projection you want**.
Different models answer different scientific questions.

Examples of models:

- **Linear models:** PCA, TICA, SVD
  - Reveal variance or slow modes
- **Nonlinear manifold learners:** t-SNE, UMAP, diffusion maps
  - Reveal intrinsic low-dimensional structure
- **Probabilistic models:** VAMPnets, state space models
  - Extract dynamical or kinetic information
- **Neural-network-based models:** autoencoders, VAEs
  - Learn compressed latent spaces capturing complex motions

Your choice of model answers:

- *Am I looking for variance?*
- *Am I looking for slow dynamics?*
- *Am I looking for clusters or metastable states?*
- *Am I looking for a generative representation?*

This step is **what shapes the lens** through which the system is viewed.

```
# Linear models
from sklearn.decomposition import PCA
from deeptime.decomposition import TICA
```

```
# import base64

# path = "/content/drive/MyDrive/smcb_data/vampnet.png"

# with open(path, "rb") as f:
#     im = base64.b64encode(f.read()).decode()

# markdown_img = f"![vampnet]({im})"
# print(markdown_img)
```

![vampnet](

🧩 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality-reduction technique that transforms a dataset into a new coordinate system such that the greatest variance lies on the first principal component, the second greatest variance on the second component, and so on.

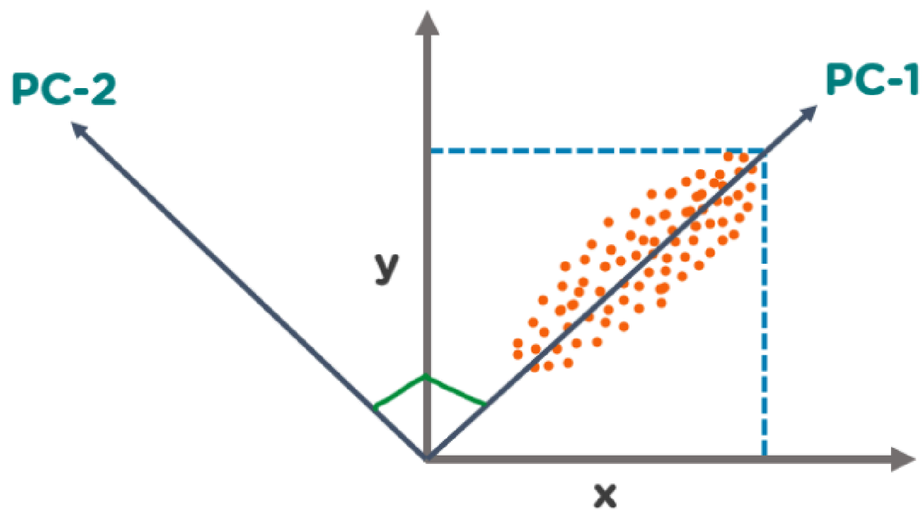
PCA is useful for:

- Visualizing high-dimensional data
- Reducing noise
- Speeding up machine learning models
- Identifying underlying structure in data

How PCA Works?

1. **Standardize the data**
2. **Compute the covariance matrix**
3. **Find eigenvalues and eigenvectors**
4. **Sort components by explained variance**
5. **Project the data onto the principal components**

📉 PCA Illustration



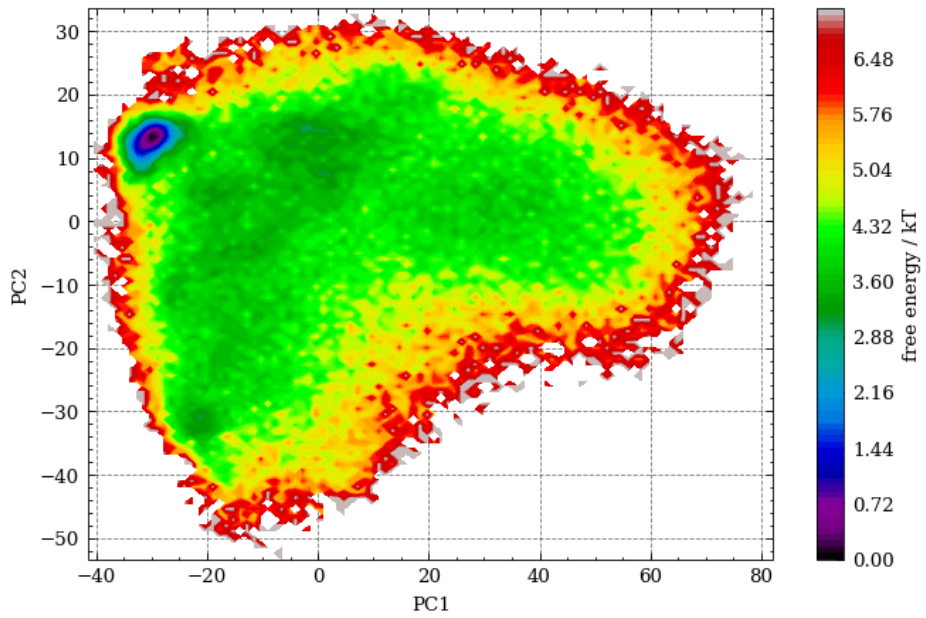
https://www.simplilearn.com/tutorials/machine-learning-tutorial/principal-component-analysis

```
pca = PCA(n_components=2)
# lagtime=1 = 2 ns
pca.fit(data)
projection_pca = pca.transform(data)
print(projection_pca)
```

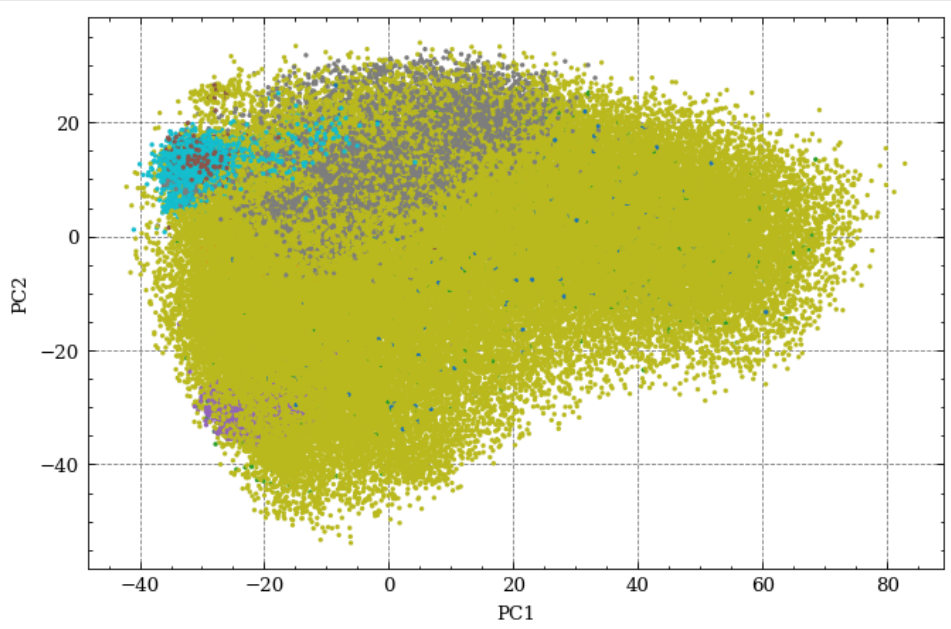
```
[[-32.12935556  13.01458514]
 [-29.70818048  12.90507133]
 [-29.45541225  15.09676999]
 ...
 [-31.11154168  13.31001061]
 [-33.00789998  12.11526852]
 [-28.08805124  14.47711716]]
```

```
# plot
plot_free_energy(projection_pca[:, 0], projection_pca[:, 1], kbt=1.0)
plt.xlabel("PC1")
plt.ylabel("PC2")
```

```
/content/drive/MyDrive/smcb_data/plot2d.py:251: UserWarning: kbt=1.0 is not an allowed optional parameter and will be ignored
  _warn(
Text(0, 0.5, 'PC2')
```




```
## Projecting along top two pc components
plt.scatter(projection_pca[:,0], projection_pca[:,1],s=1, c=color_values)
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.show()
```




```
np.corrcoef(projection_pca[:,0],rog)
```

```
array([[1.        , 0.91994003],
       [0.91994003, 1.        ]])
```

```
np.corrcoef(projection_pca[:,0],rmsd)
```

```
array([[1.        , 0.81254554],
       [0.81254554, 1.        ]])
```

⏱️ Time-lagged Independent Component Analysis (TICA)

TICA (Time-lagged Independent Component Analysis) is a linear dimensionality-reduction technique designed for time-series data. It finds linear combinations of input coordinates that are maximally autocorrelated at a specified lag time — i.e., the slow collective motions in dynamical systems (commonly used in molecular dynamics, video, and other time-series analysis).

Why use TICA?

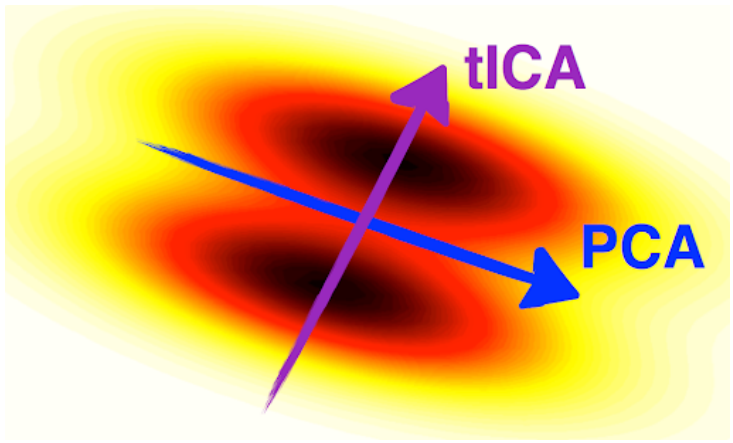
- Identifies slow, metastable processes in time-series data
- Great for preprocessing before clustering or Markov state model construction
- Reduces dimensionality while preserving slow dynamics

Algorithm (high level)

1. Standardize the data (zero mean, optionally scale).
2. Compute time-lagged covariance matrices ($C_0 = \langle x_t x_t^\top \rangle$) and ($C_\tau = \langle x_t x_{t+\tau}^\top \rangle$).
3. Solve the generalized eigenvalue problem: $C_\tau v = \lambda C_0 v$.
4. Sort eigenvectors by eigenvalue (largest → slowest processes).
5. Project trajectories onto selected time-lagged independent components (TICs).

📌 Usage notes

- Choose the lag time `tau` based on the timescales you care about (should be longer than noise, shorter than the slowest process of interest).
- Standardizing (mean-centering) per feature is usually required.
- The eigenvalues λ give the autocorrelation at lag τ; values close to 1 indicate very slow modes.

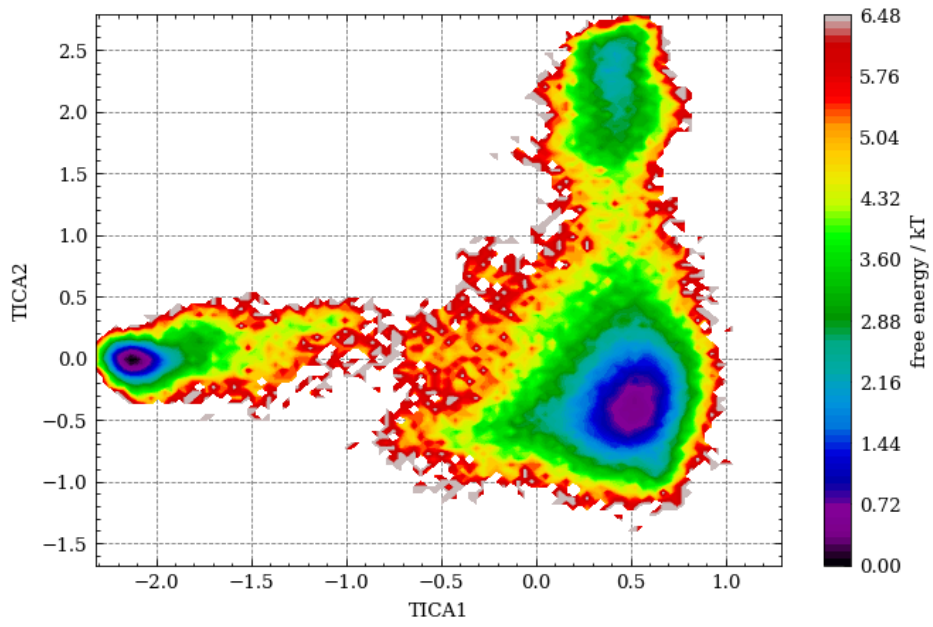


"https://www.google.com/url?sa=i&url=http%3A%2F%2Fmsmbuilder.org%2F3.3.0%2Ftica.html&psig=AOvVaw0ZOK_g5hC_CC0EbiUyOLoe&ust=1764674098273000&source=images&cd=vfe&opi=89978449&ved=0CBgQjhxqFwoTCIDYy8GhnJEDFQAAAAAdAAAAABAK"

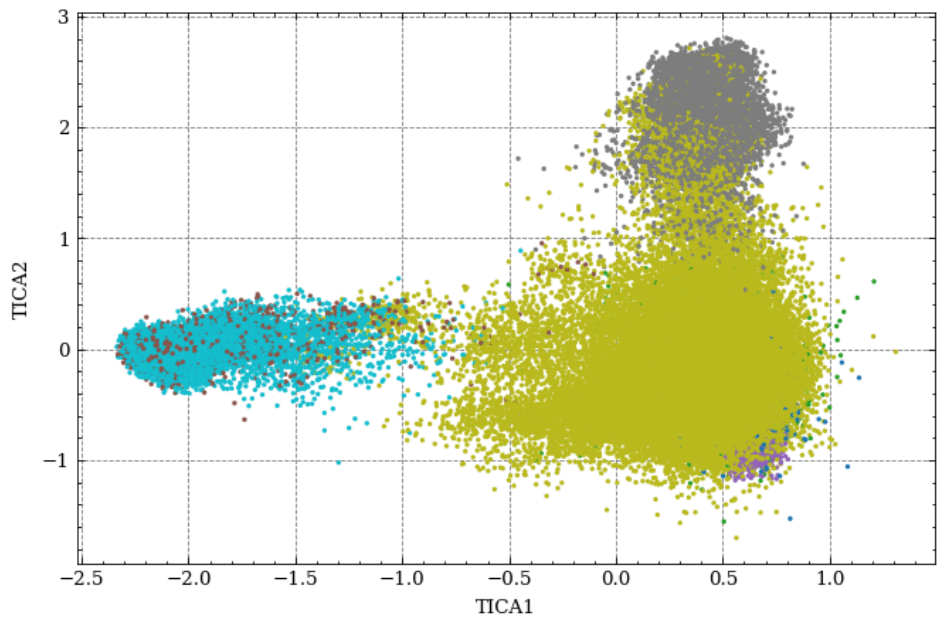
```
# lagtime = 1 = 2ns
tica = TICA(lagtime=20)

# fit TICA model
tica.fit(data)

projection_tica = tica.transform(data)
```

```
# plot
plot_free_energy(projection_tica[:, 0], projection_tica[:, 1], kbt=1.0)
plt.xlabel("TICA1")
plt.ylabel("TICA2")
```

```
Text(0, 0.5, 'TICA2')
```




```
## Projecting along top two pc components
plt.scatter(projection_tica[:,0], projection_tica[:,1],s=1, c=color_values)
plt.xlabel("TICA1")
plt.ylabel("TICA2")
plt.show()
```




```
np.corrcoef(projection_tica[:,0],q)
```

```
array([[ 1.        , -0.93599694],
       [-0.93599694,  1.        ]])
```

🔧 Autoencoders (AE) & Variational Autoencoders (VAE)

Autoencoders

Autoencoders are neural networks that learn **compressed representations** of data.
They consist of:

- **Encoder:** maps input → low-dimensional latent vector
- **Decoder:** reconstructs input from latent vector

Goal: minimize reconstruction error.

Use cases: denoising, dimensionality reduction, feature learning, anomaly detection.

Variational Autoencoders (VAE)

VAEs are probabilistic autoencoders that learn a **distribution** over the latent space instead of fixed points.

They model:

- (q_\phi(z|x) = \mathcal{N}(\mu, \sigma^2)) (encoder)
- (p_\theta(x|z)) (decoder)

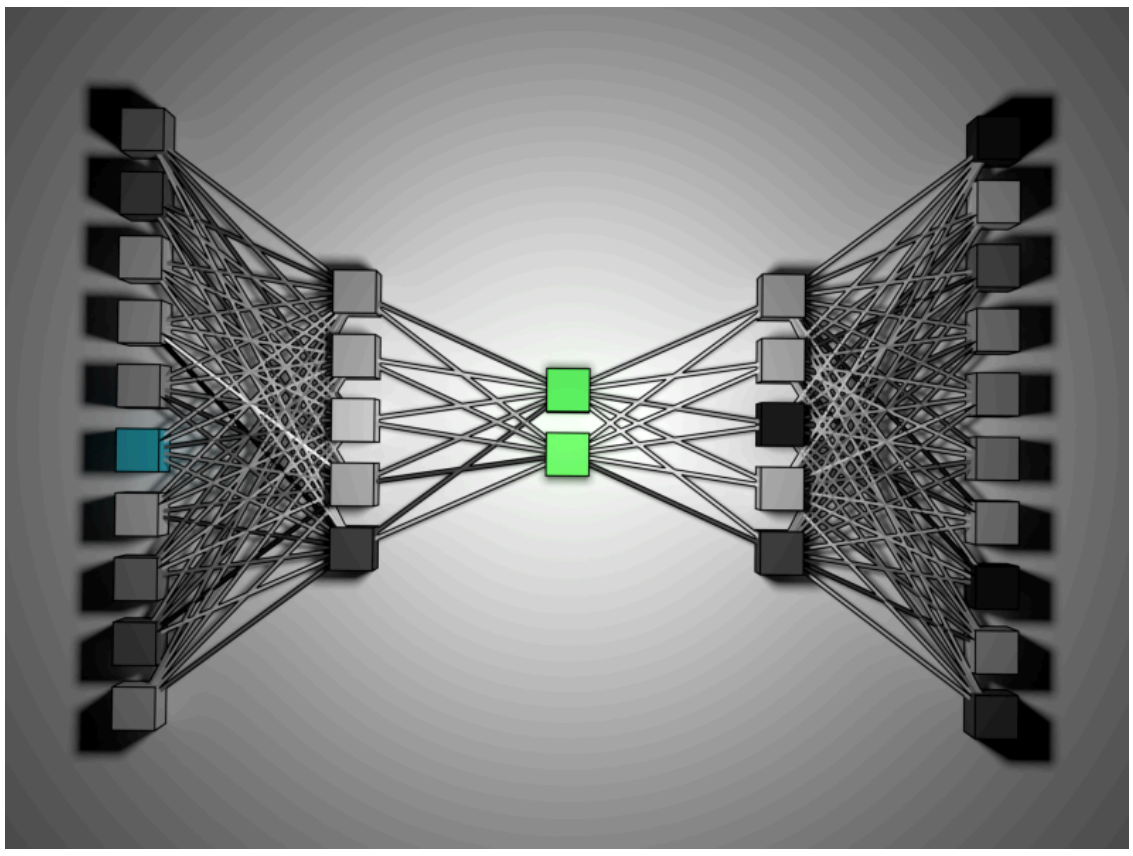
Loss = Reconstruction Loss + KL Divergence, encouraging smooth, continuous latent spaces.

Benefits:

- Generate new samples
- Smooth interpolation
- Well-structured latent manifolds

📌 Key Differences

- **AE:** deterministic, learns exact latent codes
- **VAE:** probabilistic, learns latent distributions
- **AE:** great for compression & denoising
- **VAE:** great for generative modeling



```
# Non-Linear models
import torch.nn as nn
from vae import VAE
from sklearn.model_selection import train_test_split
import torch
# ------------------------------------------------------
# Define encoder and decoder networks for VAE
# ------------------------------------------------------
input_dim = 190
latent_dim = 2

encoder = nn.Sequential(
    nn.Linear(input_dim, 32),
    nn.ReLU(),
    nn.Linear(32, 8),
    nn.ReLU(),
)

decoder = nn.Sequential(
    nn.Linear(latent_dim, 32),
    nn.ReLU(),
    nn.Linear(32, 8),
    nn.ReLU(),
    nn.Linear(8, input_dim),
)

criterion = nn.MSELoss(reduction="mean")
train_data,val_data = train_test_split(data, test_size=0.2, random_state=42)
```

```
# pip install --force-reinstall torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121
```

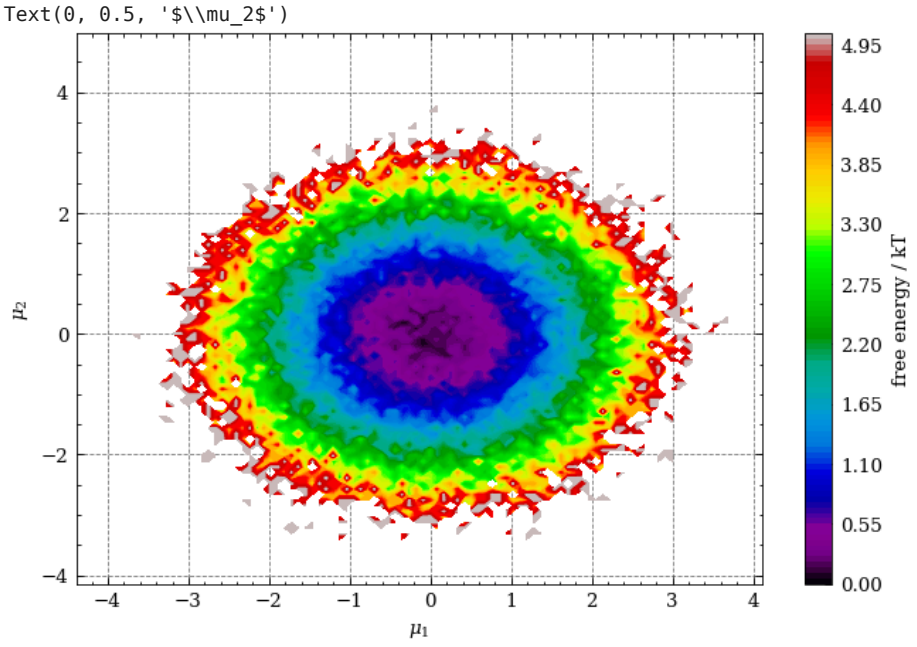
```
vae = VAE(
    input_dim=input_dim,
    latent_dim=latent_dim,
    encoder=encoder,
    decoder=decoder,
    criterion=criterion,
    train_data=train_data,
    val_data=val_data,
    beta=2.0,
    learning_rate=1e-2,
    epochs=20,
    batch_size = 128
)

vae.fit()
```

```
Epoch 1/20 | Train Loss: 27.86072 | Val Loss: 23.55609
Epoch 2/20 | Train Loss: 20.60255 | Val Loss: 17.83924
Epoch 3/20 | Train Loss: 15.97887 | Val Loss: 14.20753
Epoch 4/20 | Train Loss: 13.07584 | Val Loss: 11.97761
Epoch 5/20 | Train Loss: 11.35474 | Val Loss: 10.72688
Epoch 6/20 | Train Loss: 10.44859 | Val Loss: 10.12446
Epoch 7/20 | Train Loss: 10.05151 | Val Loss: 9.89560
Epoch 8/20 | Train Loss: 9.91771 | Val Loss: 9.83555
Epoch 9/20 | Train Loss: 9.89139 | Val Loss: 9.82749
Epoch 10/20 | Train Loss: 9.88727 | Val Loss: 9.82893
Epoch 11/20 | Train Loss: 9.88761 | Val Loss: 9.82615
Epoch 12/20 | Train Loss: 9.88712 | Val Loss: 9.82409
Epoch 13/20 | Train Loss: 9.88746 | Val Loss: 9.82596
Early stopping (no improvement)
```

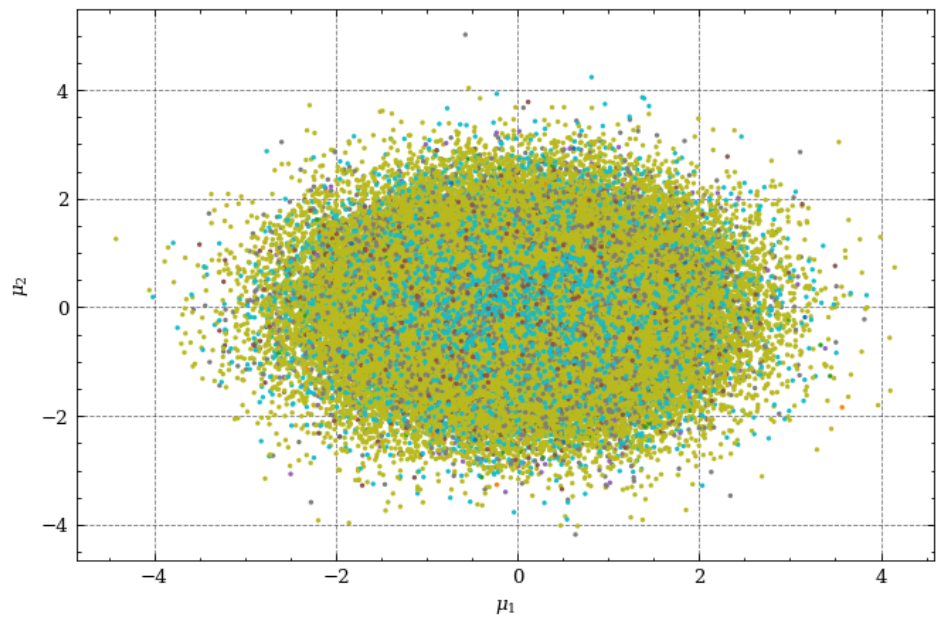
```
mu = vae.encode(
    torch.tensor(data, dtype=torch.float32).cuda()
)[0].detach().cpu().numpy()
```

```
# plot
plot_free_energy(mu[:, 0], mu[:, 1])
plt.xlabel(r"$\mu_1$")
plt.ylabel(r"$\mu_2$")
```

```
Text(0, 0.5, '$\\mu_2$')
```




```
## Projecting along top two pc components
plt.scatter(mu[:,0], mu[:,1],s=1, c=color_values)
plt.xlabel(r"$\mu_1$")
plt.ylabel(r"$\mu_2$")
plt.show()
```



🧬 Conformations as States in Molecular Dynamics

Molecular dynamics (MD) simulations explore a molecule's **conformational landscape**, where each conformation represents a **state** in a high-dimensional phase space. As trajectories evolve, the system transitions between metastable states, revealing slow collective motions that define functional behavior. Understanding these conformational states is crucial for characterizing kinetics, thermodynamics, and long-timescale biological processes.

🔁 Koopman Theory: A Linear View of Nonlinear Dynamics

Koopman operator theory provides a powerful mathematical framework for analyzing complex dynamical systems.
Although MD dynamics are **nonlinear**, the Koopman operator describes their evolution as a **linear** transformation — but in an appropriately chosen (possibly infinite-dimensional) function space.

Key ideas:

- Dynamics evolve linearly in the space of **observables**, not coordinates.
- Eigenfunctions of the Koopman operator correspond to **slow dynamical modes**.
- Approximating the Koopman operator yields a principled way to extract meaningful, long-timescale behavior from MD data.

This perspective unifies many modern dimensionality-reduction and learning techniques for MD.

📉 Dimensionality Reduction for Dynamical Systems

To approximate the **Koopman operator**, we seek **low-dimensional coordinates** capturing the system's **slow, metastable dynamics**.
Key approaches include:

• DMD (Dynamic Mode Decomposition)

Linear method extracting **temporal modes** from time-lagged snapshots. Good for coherent structures but limited for nonlinear dynamics.

• TICA (Time-lagged Independent Component Analysis)

Linear combinations of features with **maximal autocorrelation**. Can be shown the basis that maximizes the autocorrelation correspond to the eigenvectors of the koopman operator.

• TAE (Time-lagged Autoencoders)

Neural network extension of TICA. Learns **nonlinear slow modes** by reconstructing time-lagged pairs.

• VAMPnets

Deep networks trained to maximize the **VAMP score**, learning **optimal nonlinear features** that approximate the Koopman operator and reveal metastable states end-to-end.

```
from IPython.display import IFrame
IFrame("https://deeptime-ml.github.io/latest/index_dimreduction.html", width=150000, height=600)
```

Dimension reduction

Here we introduce the dimension reduction / decomposition techniques implemented in the package.

Koopman operator methods

All methods contained in this sub-package relate to the Koopman operator $\mathcal{K}_\tau$ defined as

for a process $\{x_t\}_{t\geq 0}$ with transition density $p_\tau(x, y)$. When projecting $\mathcal{K}_\tau$ into a finite basis, one seeks

where $K \in \mathbb{R}^{n\times m}$ is a finite-dimensional Koopman matrix which propagates the observable f of the system's state x_t to the observable g at state $x_{t+\tau}$.

When to use which method

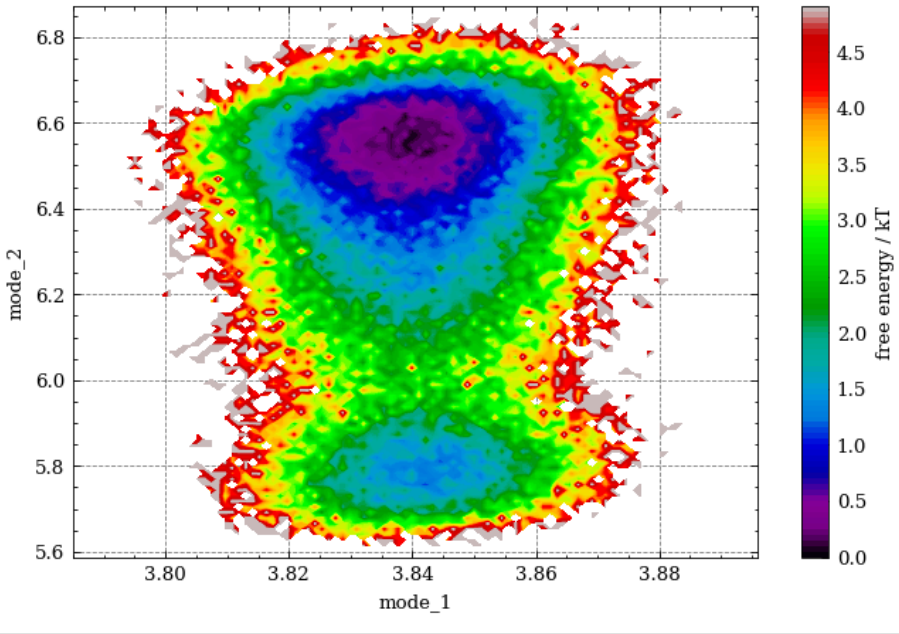
All methods assume (approximate) Markovianity of the time series under lag-time τ.

DMD

```
# Dynamic Mode decomposition
from deeptime.decomposition import DMD
dmd = DMD()
dmd.fit(data,lagtime=1)# try 150
projection_dmd = dmd.transform(data)
```

```
# plot
plot_free_energy(projection_dmd[:, 0], projection_dmd[:, 1], kbt=1.0)
plt.xlabel("mode_1")
plt.ylabel("mode_2")
```

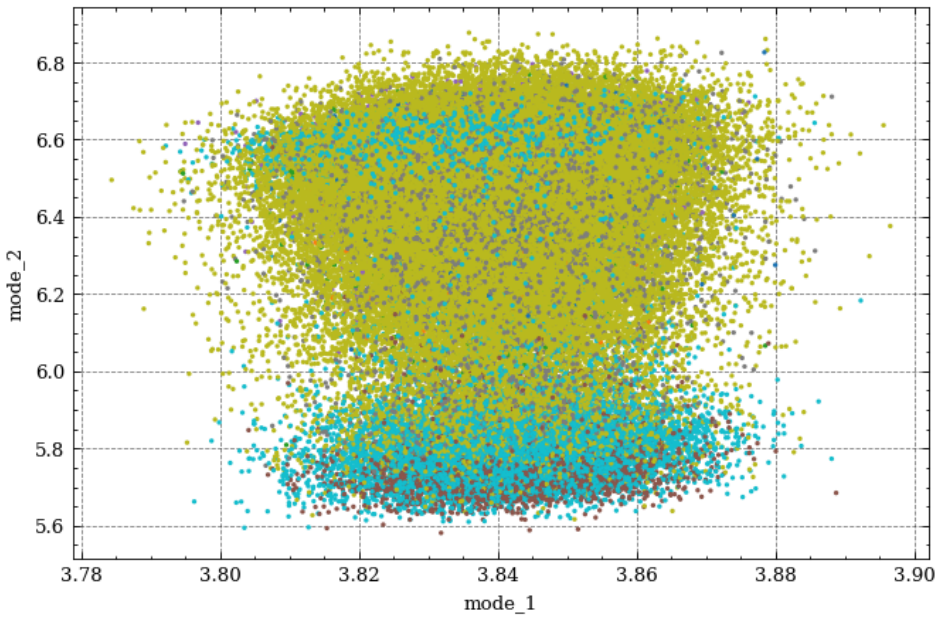
```
/content/drive/MyDrive/smcb_data/plot2d.py:251: UserWarning: kbt=1.0 is not an allowed optional parameter and will be ignored
  _warn(
/usr/local/lib/python3.12/dist-packages/matplotlib/cbook.py:1709: ComplexWarning: Casting complex values to real discards the imaginary part
  return math.isfinite(val)
/usr/local/lib/python3.12/dist-packages/matplotlib/contour.py:1387: ComplexWarning: Casting complex values to real discards the imaginary part
  x = np.asarray(x, dtype=np.float64)
/usr/local/lib/python3.12/dist-packages/matplotlib/contour.py:1388: ComplexWarning: Casting complex values to real discards the imaginary part
  y = np.asarray(y, dtype=np.float64)
Text(0, 0.5, 'mode_2')
```




```
## Projecting along top two pc components
plt.scatter(projection_dmd[:,0], projection_dmd[:,1],s=1, c=color_values)
```

```
plt.xlabel("mode_1")
plt.ylabel("mode_2")
plt.show()
```

```
/usr/local/lib/python3.12/dist-packages/matplotlib/collections.py:200: ComplexWarning: Casting complex values to real discards the imaginary part
  offsets = np.asanyarray(offsets, float)
```



TAE

```
# Time lagged Autoencoder
from deeptime.util.torch import MLP
from deeptime.decomposition.deep import TAE
from torch.utils.data import DataLoader
from deeptime.util.data import TrajectoryDataset
from tqdm.notebook import tqdm
```

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    torch.backends.cudnn.benchmark = True
else:
    device = torch.device("cpu")
```

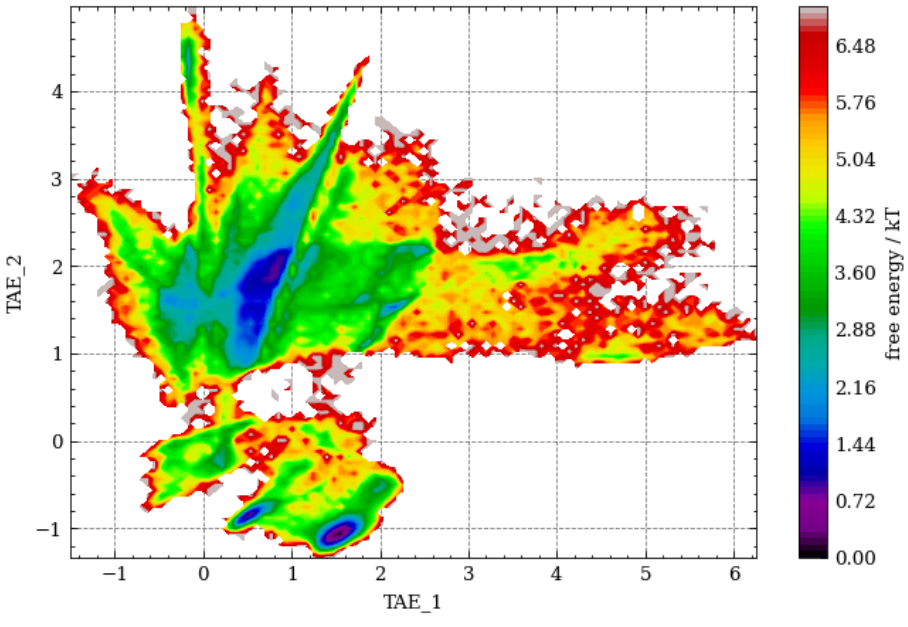
```
import joblib
# load
loaded_scaler = joblib.load("scaler.joblib")

# transform
data_scaled = loaded_scaler.transform(data)
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/base.py:380: InconsistentVersionWarning: Trying to unpickle estimator MinMaxScaler from version 1.7.2 when using
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
```

```
dataset = TrajectoryDataset(1, data_scaled.astype(np.float32)) #lagtime=1 = 2 ns
n_val = int(len(dataset)*.25)
train_data, val_data = torch.utils.data.random_split(dataset, [len(dataset) - n_val, n_val])
loader_train = DataLoader(train_data, batch_size=128, shuffle=True)
loader_val = DataLoader(val_data, batch_size=len(val_data), shuffle=False)


units = [190,1024,512, 2]
encoder = MLP(units, nonlinearity=torch.nn.LeakyReLU, output_nonlinearity=None,
              initial_batchnorm=False)
decoder = MLP(units[::-1], nonlinearity=torch.nn.LeakyReLU, initial_batchnorm=False)
tae = TAE(encoder, decoder, learning_rate=5e-4, device=device)
tae.fit(loader_train, n_epochs=100, validation_loader=loader_val,progress=tqdm)
tae_model = tae.fetch_model()
```

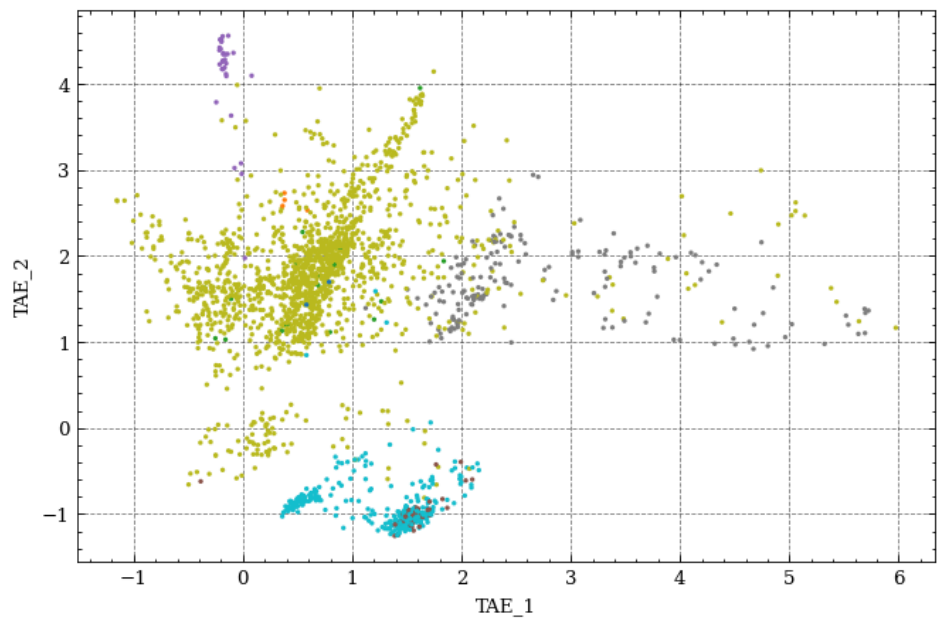
```
# plot
projection_tae = tae_model.transform(data_scaled)
plot_free_energy(projection_tae[:, 0], projection_tae[:, 1], kbt=1.0)
plt.xlabel("TAE_1")
plt.ylabel("TAE_2")
```

```
/content/drive/MyDrive/smcb_data/plot2d.py:251: UserWarning: kbt=1.0 is not an allowed optional parameter and will be ignored
  _warn(
Text(0, 0.5, 'TAE_2')
```

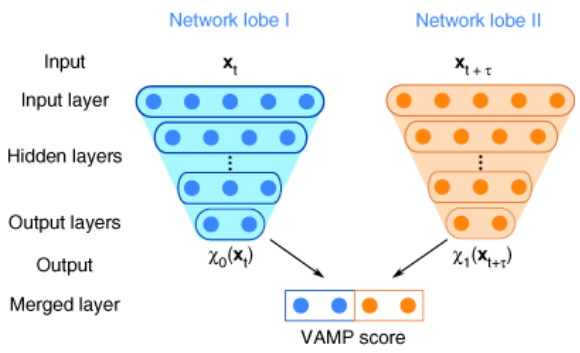



```
plt.scatter(projection_tae[::50,0], projection_tae[::50,1],s=1, c=color_values[::50])
plt.xlabel("TAE_1")
```

```
plt.ylabel("TAE_2")
plt.show()
```



VAMPNETS



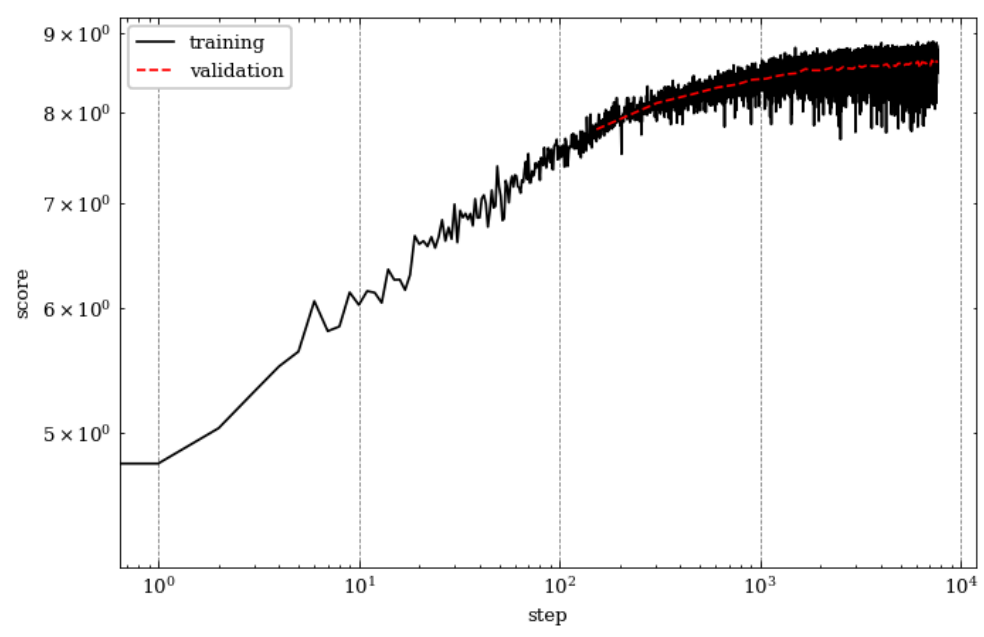
https://www.nature.com/articles/s41467-017-02388-1

```
# VAMPNETS
from deeptime.decomposition.deep import VAMPNet
dataset = TrajectoryDataset(1, data_scaled.astype(np.float32)) #lagtime=1 = 2 ns
n_val = int(len(dataset)*.25)
train_data, val_data = torch.utils.data.random_split(dataset, [len(dataset) - n_val, n_val])
loader_train = DataLoader(train_data, batch_size=512, shuffle=True)
loader_val = DataLoader(val_data, batch_size=len(val_data), shuffle=False)

lobe = MLP(units=[data.shape[1],1024, 512, 128,8], nonlinearity=nn.CELU)
lobe = lobe.to(device)

print(lobe)
```

```
MLP(
  (_sequential): Sequential(
    (0): Linear(in_features=190, out_features=1024, bias=True)
    (1): CELU(alpha=1.0)
    (2): Linear(in_features=1024, out_features=512, bias=True)
    (3): CELU(alpha=1.0)
    (4): Linear(in_features=512, out_features=128, bias=True)
    (5): CELU(alpha=1.0)
    (6): Linear(in_features=128, out_features=8, bias=True)
  )
)
```

```
vampnet = VAMPNet(lobe=lobe, learning_rate=1e-4, device=device)

model = vampnet.fit(loader_train, n_epochs=50,
                    validation_loader=loader_val, progress=tqdm).fetch_model()
```

```
plt.loglog(*vampnet.train_scores.T, label='training')
plt.loglog(*vampnet.validation_scores.T, label='validation')
plt.xlabel('step')
plt.ylabel('score')
plt.legend();
```

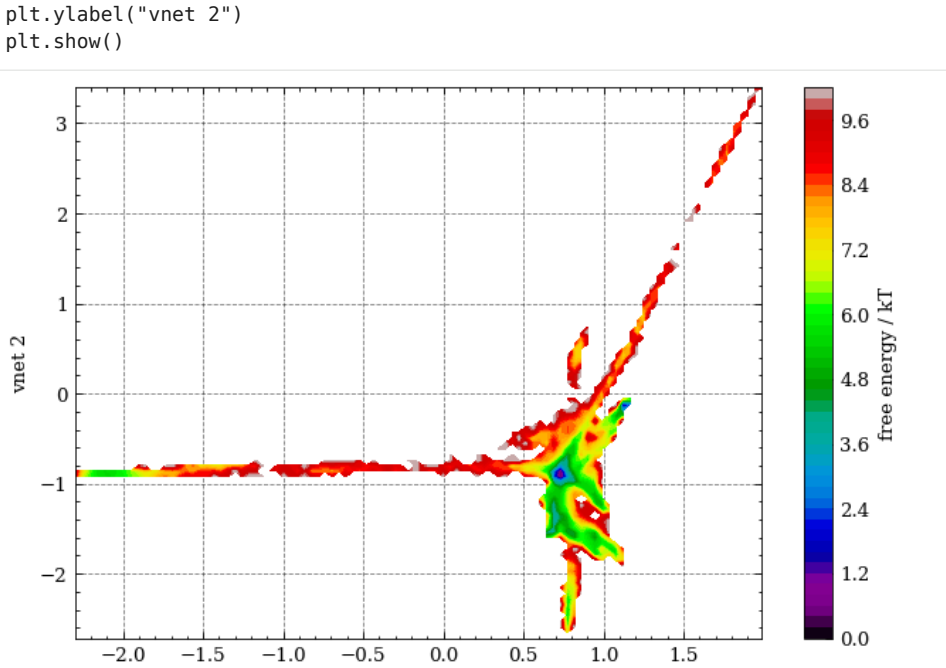



```
from deeptime.clustering import KMeans
from deeptime.markov.msm import MaximumLikelihoodMSM
```

```
# soft state probabilities
chi = model.transform(data)

dtraj_rec = KMeans(8).fit(chi).transform(chi)
msm = MaximumLikelihoodMSM().fit(dtraj_rec, lagtime=1).fetch_model()
# Transition matrix
T = msm.transition_matrix  # estimated transition matrix
tau = msm.lagtime          # lag time used in training

# Diagonalize T
import numpy as np
eigvals, eigvecs = np.linalg.eig(T)

# Sort by real part of eigenvalues
idx = np.argsort(-eigvals.real)
eigvals, eigvecs = eigvals[idx], eigvecs[:, idx]

# Compute implied timescales

tau = 1
its = -tau / np.log(np.abs(eigvals))
```

```
plt.plot(its, 'o-')
plt.xlabel('Implied state index')
plt.ylabel('Implied timescale (ns)')
plt.ylim(0,100)
```

```
(0.0, 100.0)
```




```
psi1 = chi @ eigvecs[:, 0].real
psi2 = chi @ eigvecs[:, 1].real
psi3 = chi @ eigvecs[:, 2].real
psi4 = chi @ eigvecs[:, 3].real
psi5 = chi @ eigvecs[:, 4].real
psi6 = chi @ eigvecs[:, 5].real
psi7 = chi @ eigvecs[:, 6].real
psi8 = chi @ eigvecs[:, 7].real
```

```
fig, ax = plt.subplots(figsize=(6.5, 4.2))
plot_free_energy(psi1,psi2, ax=ax, levels=50, alpha=1.0)
plt.xlabel("vnet 1")
plt.ylabel("vnet 2")
plt.show()
```



```
# Create plot
fig, ax = plt.subplots(figsize=(6.5, 4.2))

# Free energy landscape
plot_free_energy(psi2, psi7, ax=ax, levels=50, alpha=0.5,cbar=False)

# Overlay scatter colored by cluster
sc = ax.scatter(
    psi2, psi7,
    s=2,
    c=color_values,
    alpha=0.7
)

# Custom legend instead of continuous colorbar
for cl, col in cluster_to_color.items():
    ax.scatter([], [], color=col, label=f'Cluster {int(cl)}', s=10)
ax.legend(title="Clusters", bbox_to_anchor=(1.05, 1), loc='upper left')
```

```
ax.set_xlabel("vnet 1")
ax.set_ylabel("vnet 2")

plt.tight_layout()
plt.show()
```