PHYS240_Final_JacksonSheppard

June 14, 2022

1 PHYS 240: Final Project

1.1 Jackson Sheppard

[1]:
```
%%javascript
MathJax.Hub.Config({
    TeX: { equationNumbers: { autoNumber: "AMS" } }
});
```

```
<IPython.core.display.Javascript object>
```

2 Theory

2.1 Molecular Dynamics and the Weighted Histogram Analysis Method

Here we present a literature study of "BayesWHAM: A Bayesian Approach for Free Eenergy Estimation, Reweighting, and Uncertainty Quantification in the Weighted Histogram Analysis Method", by Andrew L. Ferguson. The original work can be accessed through onlinelibrary.wiley.com with the original codebase and associated documentation available through BitBucket. Updated code for the purposes of this analysis can be accessed through GitHub.

The Weighted Histogram Analysis Method (WHAM) is a technique to estimate free energy surfaces of molecular system simulated using Molecular Dynamics (MD), a technique involving the numerical integration of Newton's second law for a collection of particles, tracking positions and velocities at each time step to yield a molecular trajectory. One can then calculate thermodynamic properties as time averages over the trajectory.

MD simulations are performed using thermodynamic ensembles. Examples include the canonical NVT ensemble (constant number of particles N, volume V, and temperature T) and the isothermal-isobaric NPT ensemble (where now the pressure P is held constant in adddition to N and T). These simulation ensembles have corresponding distributions of microstates given by Boltzmann factors, e.g for the canonical ensemble:

$$P(\underline{r}^N) = \frac{e^{-\beta U(\underline{r}^N)}}{Z} \tag{1}$$

Where a given microstate is defined by the $3N$ dimension vector $\underline{r}^N$ (N particles, each defined by three coordinates), $\beta = 1/k_B T$ is the inverse temperature (where k_B is Boltzmann's constant), and $Z = \int d\underline{r}^N e^{-\beta U(\underline{r}^N)}$ is the partition function for the canonical ensemble that also serves as the normalization constant for the distribution defined in equation 1. $U(\underline{r}^N)$ then defines a potential energy surface (PES), but due to it's high dimensionality ($N \approx 10^3$ for typical simulations), this quantity can be difficult to interpret. Instead, a free energy surface (FES) is defined in terms of a set of $m < 3N$ collective order parameters $\underline{\psi}$ that serve as a low-dimensional projection of the PES, mapping multiple configurations to the same collective coordinates. This projection is achived through the dirac delta function that picks out configurations with a given value of $\underline{\psi}$:

$$P(\underline{\psi}) = \int P(\underline{r}^N)\delta(\underline{\psi}(\underline{r}^N) - \underline{\psi})d\underline{r}^N \tag{2}$$

For the canonical ensemble:

$$P(\underline{\psi}) = \frac{1}{Z}\int e^{-\beta U(\underline{r}^N)}\delta(\underline{\psi}(\underline{r}^N) - \underline{\psi})d\underline{r}^N \tag{3}$$

The free energy surface (FES) is then defined in terms of the collective order parameters as:

$$F(\underline{\psi}) = -k_B T \ln\left(P(\underline{\psi})\right) \tag{4}$$

The collective order parameters can be determined using dimensionality reduction methods such as principal component analysis, or by considering specific parameters relevant to the physics of the system of interest, e.g the dihedral bond angles ϕ and ψ of the molecular system.

The reduction of the PES to the lower dimensional FES is still insufficient to overcome high potential energy barriers typical of molecular systems, and thus unbiased MD simulations would not allow for an extensive search of the $\underline{\psi}$ parameter space. As a remedy, a technique known as umbrella sampling is employed. This process includes biasing the potential energy with typically harmonic restraining potentials of changing center locations $\underline{\psi_i}$.

$$w_i(\underline{\psi}(\underline{r}^N);\, \mathbf{K}_i, \underline{\psi_i}) = \frac{1}{2}(\underline{\psi}(\underline{r}^N) - \underline{\psi_i}))^T\, \mathbf{K}_i(\underline{\psi}(\underline{r}^N) - \underline{\psi_i})) \tag{5}$$

Here $\mathbf{K}_i$ is an m-by-m diagonal matrix of force constants in each dimension of ψ. One can then perform repeated simulations with changing center locations to complete a reaonsable exploration of parameter space in the viscinity of each ψ_i. Ferguson then shows then one can determine an estimate for the unbiased distribution of microstates in terms of the collective order parameters for the biased simulation i through the following relationship:

$$\hat{P}_i(\underline{\psi}) = \frac{n_i(\underline{\psi})}{N_i c_i(\underline{\psi}) f_i} \tag{6}$$

where $n_i(\underline{\psi})$ is the number density of observation of the system over the course of the biased simulation i in a differential volume element $d\underline{\psi}$, N_i is the total number of observations extracted in the simulation trajectory, $c_i(\underline{\psi}) = e^{-\beta w_i}$ is the Boltzmann factor of the bias potential, and $f_i = Z/Z_i^B$ is the ratio of partitition functions of the unbiased system and the biased system. The free energy is then related to the partition function by $-\beta F = \ln Z$, so we can write this ratio as

$f_i = e^{-\beta(F-F_i^B)}$ making it clear that this is the shift in free energy between the biased and unbiased systems.

We thus have an estimate of the unbiased FES by application of equation 4, i.e

$$\hat{F}_i(\underline{\psi}) = -k_B T \ln \hat{P}_i(\underline{\psi}) \tag{7}$$

2.2 Bayesian Approach to WHAM

It is thus clear that a typical umbrella sampling data set for which a WHAM analysis method could be employed would include a series of MD simulations indexed $i = 1, ...S$, each with a different bias potential w_i, and corresponding trajectories of collective order parameters $\underline{\psi}$. One can then view equation 6 as a relationship between the biased distribution $p_{i,l}^B = n_{i,l}/N_i$ (we now include index l to distinguish from the simulation index i, where l is a bin number of the $\underline{\psi}$ coordinates discretized into M bins, i.e $l = 1, ..., M$) and the unbiased distribution $\hat{p}_{i,l}$:

$$\hat{p}_{i,l} = \frac{p_{i,l}^B}{c_{i,l} f_i} \tag{8}$$

where $c_{i,l}$ is the Boltzmann factor of the bias potential in bin l of simulation i. The Bayesian approach to this problem is then to find the best unbiased distribution $\{p_l\}$ (the model) given our biased simulation bin counts $\{n_{i,l}\}$ (the data). This inverts equation 8 to be viewed as a relationship between the true unbiased distribution $\{p_l\}$ and a corresponding predicted biased distribution $\hat{p}_{i,l}^B$:

$$\hat{p}_{i,l}^B = p_l c_{i,l} f_i \tag{9}$$

We thus proceed using Bayes' Theorem:

$$P(\{p_l\}|\{n_{i,l}\}) = \frac{P(\{n_{i,l}\}|\{p_l\})P(\{p_l\})}{P(\{n_{i,l}\})} \tag{10}$$

$P(\{p_l\}|\{n_{i,l}\})$ is referred to as the posterior probability of the unbiased distribution $\{p_l\}$ given the data $\{n_{i,l}\}$, $P(\{n_{i,l}\}|\{p_l\})$ is the probability of the data given the model (the likelihood), $P(\{p_l\})$ is the prior probability of our model before observing data, and $P(\{n_{i,l}\})$ is the probability of observing our particular data set. $P(\{n_{i,l}\})$ is referred to as the evidence but can be interpreted as a normalization factor of the posterior, i.e $P(\{n_{i,l}\}) = \int P(\{n_{i,l}\}|\{p_l\})P(\{p_l\})d\{p_l\}$. We then determine our best model by maximization of this posterior distribution. We thus see that a uniform prior ($P(\{p_l\})$ = C), would cancel through this normalization, and the posterior distribution would be the same as the normalized likelihood, and thus a maximum posterior estimate of the unbiased distribution would be the same as a maximum likelihood estimate. This however does not need to be the case, and Ferguson instead develops maximum posterior estimates for arbitrary priors.

The task is now to express the likelihood function of observing the data (biased bin counts) given the true unbiased distribution $\{p_l\}$. Ferguson does so by considering the likelihood for a single simulation i as a multinomial distribution, i.e:

$$\mathcal{L}_i = P_i(n_{i,1}, n_{i,2}, ..., n_{i,M} | \{p_l\}) = \frac{N_i!}{\prod_{l=1}^{M} n_{i,l}!} \prod_{l=1}^{M} (\hat{p}^B_{i,l})^{p^B_{i,l} N_i} = \frac{(\sum_{l=1}^{M} n_{i,l})!}{\prod_{l=1}^{M} n_{i,l}!} \prod_{l=1}^{M} (p_l c_{i,l} f_i)^{n_{i,l}} \tag{11}$$

where the last equality follows from equation 9 and the fact that $N_i = \sum_{l=1}^{M} n_{i,l}$. Assuming each biased simulation is independent, the total likelihood is then given by the product over the S simulations of the individual likelihoods $\mathcal{L}_i$. Ferguson simplifies this expression and finds:

$$\mathcal{L} = P(\{n_{i,l}\} | \{p_l\}) = D(\{n_{i,l}\}, \{c_{i,l}\}) \prod_{l=1}^{M} p_l^{M_l} \prod_{i=1}^{S} \left(\sum_{l=1}^{M} p_l c_{i,l} \right)^{-N_i} \tag{12}$$

where $D(\{n_{i,l}\}, \{c_{i,l}\})$ is a multiplicative factor that importantly does not depend on the desired unbiased distribution p_l, $N_i = \sum_{l=1}^{M} n_{i,l}$ is the total number of observations in simulation i aggregated over all M bins, and $M_l = \sum_{i=1}^{S} n_{i,1}$ is the total number of observations in bin l aggregated over all S simulations.

2.3 Maximum Posterior/Maximum Likelihood Estimates of the Unbiased FES

Equipped with the likelihood function, Ferguson develops a best estimate for the true unbiased distribution by maximization of the posterior distribution defined in equation 10 with respect to the unbiased distribution $\{p_l\}$. As stated previously, this is equivalent to maximizing the likelihood function itself in the case of a uniform prior, but Ferguson performs this maximization for arbitrary priors using the method of Langrange multipliers. This yields the following estimate of the distribution of unbiased counts $\{p_l\}^{MAP}$:

$$p_k = \frac{M_k + p_k \frac{\partial \ln P(\{p_l\})}{\partial p_k}|_{p_{l \neq k}}}{\sum_{i=1}^{S} N_i c_{i,k} f_i + \sum_{k=1}^{M} p_k \frac{\partial \ln P(\{p_l\})}{\partial p_k}|_{p_{l \neq k}}}, \quad k = 1...M \tag{13}$$

$$f_i^{-1} = \sum_{l=1}^{M} p_l c_{i,l}, \quad i = 1...S \tag{14}$$

where equation 14 follows from normalization of the biased distribution of bin counts. This then leads to a best estimate of the unbiased FES over the discretized collective coordinates, $\{F_l\}^{MAP}$:

$$F_k = -k_B T \ln\left(\frac{p_k^{MAP}}{V_k}\right) + C, \quad k = 1...M \tag{15}$$

where V_k is the volume of bin k in the collective parameters and C is an arbitrary constant corresponding to the fact that only energy differences are meaningful. Given a data set of umbrella sampled MD simulations, one can then solve equation 13 for the distribution of unbiased counts by iteration of p_k until self-consistency is achieved. We note however that in the case of a uniform prior where the maximum posterior distribution estimate is equivalent to the maximum likelihood estimate and $\partial \ln P(\{p_l\})/\partial p_k = 0$, equation 13 reduces to that for $\{p_l\}^{ML}$:

$$p_k = \frac{M_k}{\sum_{i=1}^{S} N_i c_{i,k} f_i}, \quad k = 1...M \tag{16}$$

Thus for a uniform prior, no iteration is necessary and the unbiased distribution can simply be calculated from the biased data set. Ferguson then describes the iterative process to solve equation 13 for an arbitrary prior in Algorithm 1 of his work, summarized below in a Pythonic formulation:

```
def BayesWHAM(data, bias_boltz_fact, prior, tol):
    """
    data : {n_{i, l}}, Biased simulation counts
    bias_boltz_fact : {c_{i, l}}, Bias potential boltzmann fatcors
    prior : functional form of the prior distribution
    tol : self-consistency tolerance
    """
    # Initialization
    M_k = np.sum(data[i])  # Sum over i, do so for k = 1...M
    N_i = np.sum(data[k])  # Sum over k (= l), do so for i = 1...S
    p_k = 1/M  # k = 1...M, initial guess for unbiased distribution = uniform
    f_i = np.sum(p_k * bias_boltz_fact[k])**-1  # Sum over k (= l), do so for i = 1...S
    dp_max = float("Inf")  # Large value for initial deviation from self-consistency

    while dp_max > tol:
        # Save the old distribution
        p_k_old = p_k
        # If uniform prior, no need to iterate, just solve for the best distribution
        if prior = None:
            p_k = M_k/np.sum(N_i * bias_boltz_fact[i] * f_i)  # Sum over i, do so for k = 1...M
        else:
            # Otherwise we need to iterate by applying the BayesWHAM equation
            dp_max_inner = float("Inf")
            p_k_inner = p_k
            while dp_max_inner > tol:
                p_k_inner_old = p_k_inner
                p_k_inner = BayesWHAM(p_k_inner)
                dp_max_inner = max(p_k_inner - p_k_inner_old)  # max over k
            p_k = p_k_inner
        # Renormalize the distribution
        p_k = p_k/np.sum(p_k)
        # Recalculate f_i
        f_i = np.sum(p_k * bias_boltz_fact[k])  # Sum over k (= l), do so for i = 1...S
        # Recalculate dp_max
        dp_max = max(p_k - p_k_old)  # max over k

    return p_k
```

2.4 Uncertainty Quantification by Markov Chain Monte Carlo

Ferguson continues his analysis to formulate a routine to calculate the uncertainty in the FES predicted by maximization of the posterior distribution using the Metropolis-Hastings (MH) implementation of Markov Chain Monte Carlo (MCMC), in which a sequence of realizations of the unbiased distribution $\{p_l\}$ is generated as a Markov Chain for which statistics can be calculated from. These realizations are condisered samples of the true posterior distribution. Enforcing that one accept/reject proposal moves in the distribution of $\{p_l\}$ according to the rule of detailed balance, Ferguson derives the following acceptence criterion for the random walk in $\{p_l\}$-space employed by the Metropolis-Hastings algorithm:

$$\alpha(\{p_l\}^\nu|\{p_l\}^\mu) = \min\Big[\exp\big(\sum_{l=1}^{M} M_l \ln(\frac{p_l^\nu}{p_l^\mu}) + \sum_{i=1}^{S} N_i \ln(\frac{f_i^\nu}{f_i^\mu}) + \ln(\frac{P(\{p_l\}^\nu)}{P(\{p_l\}^\mu)})\big), 1\Big] \qquad (17)$$

where ν is the proposed state and μ is the current state of the random walk. We again note that the last term of exponential in equation 17 vanishes in the case of a uniform prior. Given this acceptance criteria, one can generate a series of unbiased distributions $\{p_{l,q}\}^{MH}$, where the index q now indicates the MCMC step number. One can then determine a sequence of free energy curves $\{F_{l,q}\}^{MH}$, each generated from a different sample of the true posterior distribution of unbiased counts:

$$F_{k,q} = -k_B T \ln(\frac{p_k^{MH}}{V_k}) + C_q, \quad k = 1...M \qquad (18)$$

Having expressed the acceptence criterion in terms of biased trajectory parameters and the unbiased distribution at the current and proposed Markov Chain steps, Ferguson describes a procedure to store the posterior sampled distributions in Algorithm 2 of his work, again reformulated as Pythonic pseudo-code:

```
def BayesMH(data, bias_boltz_fact, prior, guess, nMCMC, dp)
    """
    data : {n_{i, l}}, Biased simulation counts
    bias_boltz_fact : {c_{i, l}}, Bias potential boltzmann fatcors
    prior : functional form of the prior distribution
    guess : initial guess for unbiased distribution -> maximum posterior result p_k^MAP
    nMCMC : number of Markov Chain Monte Carlo steps to conduct
    dp : maximum proposal step size
    """
    # Initialization
    M_k = np.sum(data[i])  # Sum over i, do so for k = 1...M
    N_i = np.sum(data[k])  # Sum over k (= l), do so for i = 1...S
    p_k = guess  # k = 1...M, initial guess for unbiased distribution = uniform
    f_i = np.sum(p_k * bias_boltz_fact[k])**-1  # Sum over k (= l), do so for i = 1...S

    p_k_MH = np.zeros(nMCMC)
    for q in range(nMCMC):
        # Save the current state
        p_k_prime = p_k
        f_i_prime = f_i
        # Select a random bin to maintain symmetry required for detailed balance
        bin = np.randint(1, M)
        # Symmetric random walk trial move:
        p_k[bin] = p_k[bin] + (2*np.rand(0, 1) - 0.5)*dp
        # Renormalize the resulting distribution
        p_k = p_k/np.sum(p_k)
        # Update f_i
        f_i = np.sum(p_k * bias_boltz_fact[k])**-1
        # Calculate likelihood ratio:
        r = AcceptanceCriterionExp(p_k, p_k_prime, f_i, f_i_prime, prior(p_k), prior(p_k_prime))
        if randFloat(0, 1) > r:
            # Reject move and revert to the previous state
            p_k = p_k_prime
            f_i = f_i_prime
        # Store updated chain
        p_k_MH[q] = p_k
        return p_k_MH
```

This algorithm results in a two dimensional array $\{p_{l,q}\}^{MH}$ holding the unbiased distribution of counts in bin l of the discretized collective parameters for step q of the Markov Chain, and equation 18 gives the resulting free energy surfaces sampled from their posterior distribution. We visualize the progression of this chain by plotting the log of the sampled posterior distribution (log likelihood in the case of a uniform prior) versus the its MCMC step. After discarding the burn-in time, this value will fluctuate about a steady value, at which point the sampled posterior distribution has converged to the true posterior. One can then compute correlation lengths and infer uncertainties from the statistics of this resulting distribution.

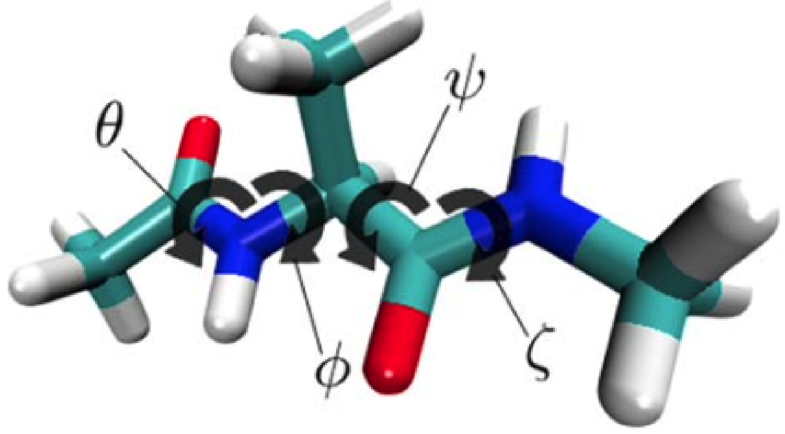
Finally, in order to visually compare the FES pedicted by a single maximum posterior estimate and the distribution of those predicted by MCMC, the additive constants C and C_q need to be set such that the landscapes predicted by each method are aligned. Ferguson does this by setting each additive constant such that it's corresponding FES has zero mean, i.e:

$$C: \quad \sum_{l=1}^{M} F_l^{MAP} = 0, \qquad C_q: \quad \sum_{l=1}^{M} F_{l,q}^{MH} = 0 \tag{19}$$

3 Application

3.1 Alanine dipeptide

We now consider applications of the previously described Bayesian reformulation of the WHAM analysis method to umbrella sampled MD trajectories, beginning with the data set included in Ferguson's original codebase, `diala_phi_EXAMPLE`. This data set corresponds to the trajectory of alanine dipeptide umbrella sampled with a one dimensional collective order parameter: the dihedral bond angle ϕ. Alanine dipeptide is a typical system system used in molecular simulations to model dihedral angles, torsion angles along a polypeptide main chain defined by four sequentially bonded atoms. Any dihedral angle is thus defined by the angle between two planes, both of which pass through the same central bond, but one of which is aligned with the next side-chain along the given central bond. The dihedral ϕ of a polypeptide is then defined by setting the central bond to that between the amide nitrogen of residue i and the alpha carbon of residue i and the side chain to that subtended by the acyl carbon of residue i to that of of residue $i-1$, where residues correspond to amino acids. (Mironov et. al). The dihedral angles are also of interest here since they are experimentally measureable from X-ray crystallography experiments through the use of Ramachandra plots. Ferguson's work provides a visualization of these angles for alanine dipeptide, included below:



Returning to the data set, input files include `diala_phi_EXAMPLE/harmonic_biases.txt` which includes 18 bias potentials specified by their index, central location of the bias ϕ angle in degrees, and the force constant used for this harmonic bias potential in kJ/(mol deg^2). For these simulations, the bias locations were set to $\phi_c = [-170°, -150°, ..., 150°, 170°]$, each with a force constant of 0.03046 kJ/(mol deg^2). We then have `diala_phi_EXAMPLE/hist/hist_binEdges.txt` which defines the M bins used in the discretization of our collective parameter ϕ. Here the ϕ coordinate has been discretized using 5° increments, i.e $\phi = [-180°, -175°, ..., 175°, 180°]$. Ferguson then performed biased MD simulations using the Gromacs 4 simulation package and generated 18 biased trajectories of the dihedral ϕ angle, the data for which is found in `diala_phi_EXAMPLE/traj`. He then compiles the biased counts (our data for this Bayesian analysis) in each bin defined in `hist_binEdges.txt` for each simulation in files `diala_phi_EXAMPLE/hist/hist_i.txt`, with i ranging from 1 to 18.

As an initial benchmark, we run Ferguson's analysis scripts `BayesWHAM.py` to generate the a free energy surface predicted from by the maximum posterior distribution of these surfaces. We then quantify uncertainties in this estimate using the previously described MCMC procedure to generate realizations of the FES from its posterior, and plot the time series of the log likelihood for over the realizations to determine its steady state. We do so using a simple `bash` script `run_bayeswham.sh` that defines input parameters, activates the necessary `conda` environment, and calls `BayesWHAM.py`. Input parameters include the dimensionality of our collective parameters (in this case 1), a periodicty flag indicating that these simulations were performed using periodic boundary condition with period set to the range of the histogram bins, and the following numerical parameters:

Parameter	Description	Value
`T`	Temperature (K)	`298`
`tol_WHAM`	Tolerance in iterative solution for $\{p_l\}^{MAP}$	`1e-15`

Parameter	Description	Value
maxIter_WHAM	Maximum number of iterations in solution for $\{p_l\}^{MAP}$ if non-convergent	1e6
steps_MH	Number of MH steps	1e6
saveMod_MH	Save frequency of MH steps	1e3
printMod_MH	Print frequency of MH steps	1e3
maxDpStep_MH	Maximum step size in MH steps, tune to achieve a ~10-50% acceptence ratio	5e-4

Finally, we include a random seed to generate reproducible results, and set the `prior` flag to `"none"` to consider a uniform prior distribution. For this reason, the maximum posterior distribution of unbiased counts is identical to the maximum likelihood distribution, and the log posterior is simply the log likelihood.

[104]:
```
# Imports
import corner
from emcee import ptsampler
import matplotlib.pyplot as plt
import numpy as np
import os
from scipy.signal import argrelextrema
import subprocess

# Use Tim's style guide
plt.style.use("config/phys240.mplstyle")
```

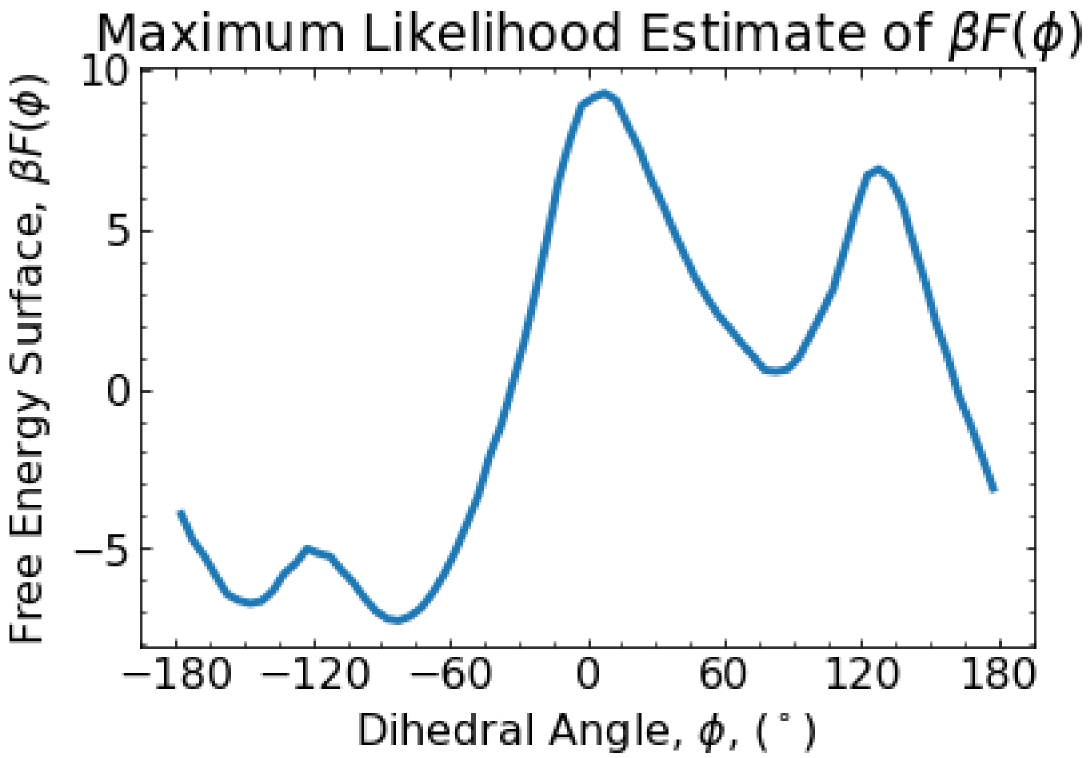
[3]:
```
# Bash script to Ferguson's original analysis scripts (written in python2.7)
subprocess.run(["./run_bayeswham.sh"])

# Clean up output files
subprocess.run("mv *.txt data_out_uniformPrior/", shell=True)
```

[3]:
```
CompletedProcess(args='mv *.txt data_out_uniformPrior/', returncode=0)
```

[4]:
```
# Create plot of unbiased free energy landscape (FES)
# First get binCenters
binC = []
f__hist_binCenters = os.path.join("data_out_uniformPrior", "hist_binCenters.txt")
with open(f__hist_binCenters, "r") as fin:
    for line in fin:
        binC.append(line.strip().split())
binC = [[float(y) for y in x] for x in binC]  # note this process scales with␣
 ↪higher dimensional collective params
binC = np.array(binC)
```

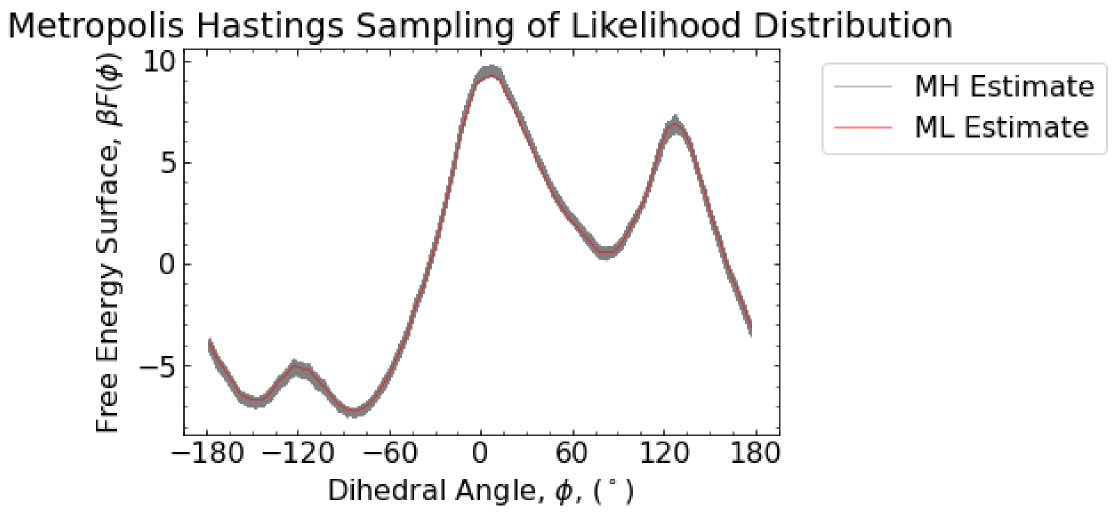
[5]:
```
# Load betaF_MAP estimate
f__betaF_MAP = os.path.join("data_out_uniformPrior", "betaF_MAP.txt")
fin = open(f__betaF_MAP, "r")
line = fin.readline()
betaF_MAP = line.strip().split()
betaF_MAP = [float(x) for x in betaF_MAP]
betaF_MAP = np.array(betaF_MAP)
fin.close()
```

[6]:
```
# Plot betaF_MAP vs binCenters (phi)
plt.plot(binC[0], betaF_MAP)
plt.xlabel(r"Dihedral Angle, $\phi$, $(^\circ)$")
plt.xticks(np.arange(-180, 181, 60))
plt.ylabel(r"Free Energy Surface, $\beta F(\phi)$")
plt.title(r"Maximum Likelihood Estimate of $\beta F(\phi)$")
plt.show()
```



[7]:
```
f__betaF_MH = os.path.join("data_out_uniformPrior", "betaF_MH.txt")

betaF_MH = []
with open(f__betaF_MH, "r") as fin:
    for line in fin:
        betaF_MH.append(line.strip().split())
betaF_MH = [[float(y) for y in x] for x in betaF_MH]
betaF_MH = np.array(betaF_MH)
```

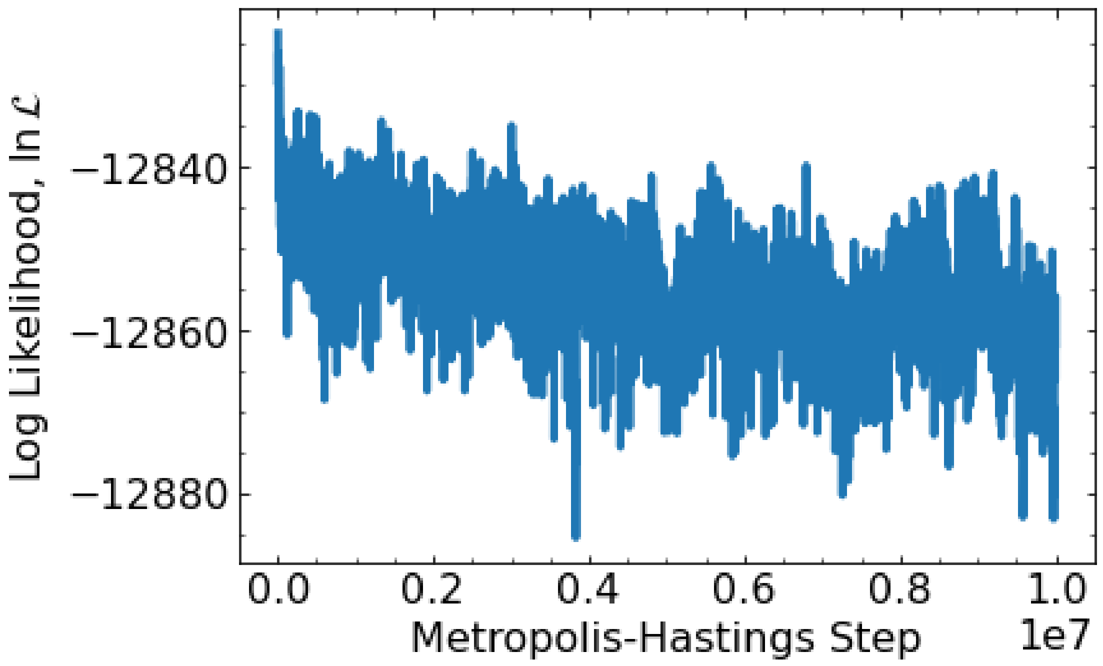
[8]:
```
# Plot betaF_MH -> Realizations of the full posterior distribution
nSamples_MH = betaF_MH.shape[0]
for k in range(0,nSamples_MH):
    if k == 0:
        plt.plot(binC[0], betaF_MH[k, :], color="0.5", lw=0.5, label="MH␣
 ↪Estimate")
    else:
        plt.plot(binC[0], betaF_MH[k, :], color="0.5", lw=0.5)
plt.plot(binC[0], betaF_MAP, "r", lw=0.5, label="ML Estimate")
plt.xlabel(r"Dihedral Angle, $\phi$, $(^\circ)$")
plt.ylabel(r"Free Energy Surface, $\beta F(\phi)$")
plt.title("Metropolis Hastings Sampling of Likelihood Distribution")
plt.xticks(np.arange(-180, 181, 60))
plt.legend(bbox_to_anchor=(1.05, 1.0), loc="upper left")
plt.show()
```



[9]:
```
# Load log likelihood vs MH step and plot
step_MH = []
f__step_MH = os.path.join("data_out_uniformPrior", "step_MH.txt")
with open(f__step_MH, "r") as fin:
    for line in fin:
        step_MH.append(int(line))
step_MH = np.array(step_MH)

logL_MH = []
```

```
f__logL_MH = os.path.join("data_out_uniformPrior", "logL_MH.txt")
with open(f__logL_MH, "r") as fin:
    for line in fin:
        logL_MH.append(float(line))
logL_MH = np.array(logL_MH)
```

[10]:
```
# Plot logL_MH vs step_MH
plt.plot(step_MH, logL_MH)
plt.xlabel("Metropolis-Hastings Step")
plt.ylabel(r"Log Likelihood, $\ln{\mathcal{L}}$")
plt.show()
```



In order to compute statistics on our Markov Chain of free energy surfaces, we must first discard the burn-in and compute the correlation length. From the time series plot of the log likelihood, we see oscillation about a log likelihood of roughly -12860 by step 4×10^6 of our Markov chain, so we set the burn-in time as:

$$n_{burn} = 4 \times 10^6 \text{ steps}$$

After discarding the burn-in we can compute the correlation length in two ways. First, we can divide our chain into many segments of length n steps. We then define the correlation length n_{corr} as the n which satisfies:

$$\frac{1}{2}\sigma_c^2 = \langle \sigma_n^2 \rangle$$

Where σ_c^2 is the variance of the entire chain and $\langle \sigma_n^2 \rangle$ is the average variance over the subchains of length n. Next, we can estimate the correlation length using the autocorrelation function:

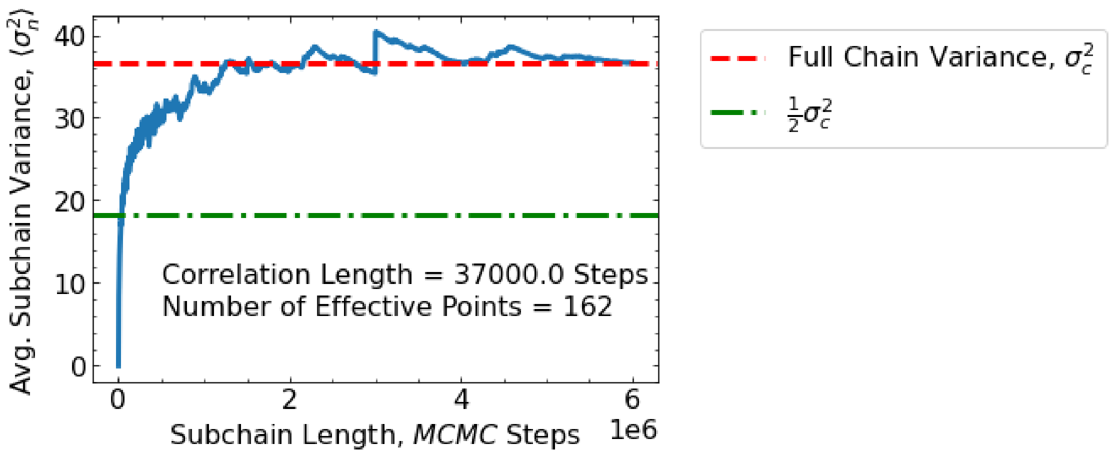
$$f(x,n) = \frac{1}{N} \sum_{i=1}^{N} \frac{(x_{i+n} - x_i)^2}{\langle x^2 \rangle - \langle x \rangle^2}$$

We then take the correlation length to be the n for which $f(x,n) = 1$. With an estimate of the correlation length, we can store every n_{corr}^{th} step of our chain resulting in a collection of independent and identically distributed draws from the posterior distribution for which we can perform statistics (a process known as chain thinning).

[11]:
```
# Discard burn-in
SAVE_MOD_MH = 1e3
STEPS_MH = 1e7
n_burn = .4e7
n_burn_idx = int(n_burn/SAVE_MOD_MH - 1)
step_MH_burned = step_MH[n_burn_idx:]
logL_MH_burned = logL_MH[n_burn_idx:]

# Calculate Correlation length: first by dividing chain into subchains of length␣
↪n
def find_corr_length(param, n_vals):
    var_subchain_avgs = np.zeros(len(n_vals))
    for i in range(len(n_vals)):
        # Need to split evenly, get remainder of total burned chaiend
        # divided by n, remove this many elements from the front
        remain = len(param) % n_vals[i]
        subchains = np.asarray(np.split(param[remain:], len(param[remain:]) //␣
↪n_vals[i]))
        subchain_vars = np.var(subchains, axis=1)
        var_subchain_avgs[i] = np.mean(subchain_vars)
    # Find correlation length
    corr_length_idx = (np.abs(var_subchain_avgs - tot_var/2)).argmin()
    return (corr_length_idx, var_subchain_avgs)

n_vals = np.arange(1, len(step_MH_burned))
tot_var = np.var(logL_MH_burned)

# Find correlation length and effective number of points
corr_length_idx, var_subchain_avgs = find_corr_length(logL_MH_burned, n_vals)
corr_length = n_vals[corr_length_idx]*SAVE_MOD_MH
N_eff = (STEPS_MH - n_burn)/corr_length
print("Subchain Variances:")
print("Correlation Length (MCMC Steps):", corr_length)
print("Number of Effective Points:", N_eff)
print("(1/2)Var(Full Chain):", tot_var/2)
print("<Var(Subchains)> at Correlation Length:",␣
↪var_subchain_avgs[corr_length_idx])
```

```
Subchain Variances:
Correlation Length (MCMC Steps): 37000.0
Number of Effective Points: 162.16216216216216
(1/2)Var(Full Chain): 18.335727764752665
<Var(Subchains)> at Correlation Length: 18.32024601177756
```

[12]:
```
# Subchain variance plot
f, ax = plt.subplots()
ax.plot(n_vals*SAVE_MOD_MH, var_subchain_avgs)
ax.axhline(y=tot_var, color='r', linestyle='--', label=r"Full Chain Variance,␣
↪$\sigma_c^2$")
ax.axhline(y=tot_var/2, color='g', linestyle='-.',␣
↪label=r"$\frac{1}{2}\sigma_c^2$")
ax.legend(bbox_to_anchor=(1.05, 1.0), loc="upper left")
ax.set_xlabel(r"Subchain Length, $MCMC$ Steps")
ax.set_ylabel(r"Avg. Subchain Variance, $\langle \sigma_n^2 \rangle$")
ax.annotate("Correlation Length = {} Steps".format(corr_length), xy=(0.5e6, 10))
ax.annotate("Number of Effective Points = {}".format(int(N_eff)), xy=(0.5e6, 6))
plt.show()
```



[13]:
```
# Determine correlation length using autocorrelation function:
def autocorr(x, n):
    """
    x : np.array
    n : int
    """
    N = len(x)
    tot = 0
    tot_var = np.var(x)
    for i in range(0, N - n):
        tot += (x[i + n] - x[i])**2/tot_var
    result = tot/N
    return result

autocorr_vals = np.zeros(len(n_vals))
for i in range(len(n_vals)):
    autocorr_n = autocorr(logL_MH_burned, n_vals[i])
    autocorr_vals[i] = autocorr_n
```

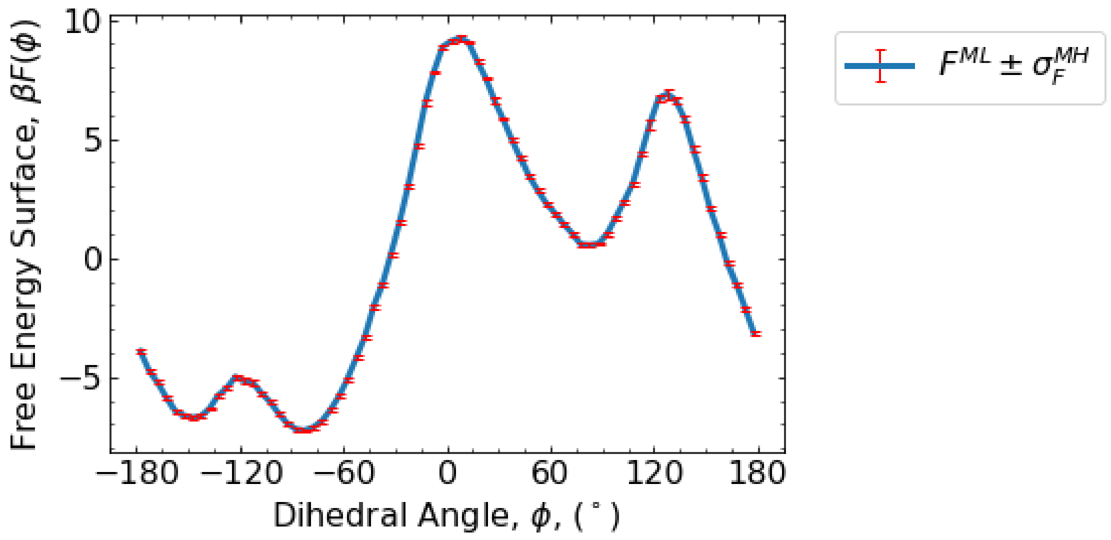
```
[14]: # Find n for which f(x, n) = 1
autocorr_len_idx = (np.abs(autocorr_vals - 1)).argmin()
autocorr_len = n_vals[autocorr_len_idx]*SAVE_MOD_MH
autocorr_N_eff = (STEPS_MH - n_burn)/autocorr_len
print("Autocorrelation Function:")
print("Correlation Length", autocorr_len)
print("Number of Effective Points:", autocorr_N_eff)
print("f(x, n_corr):", autocorr_vals[autocorr_len_idx])
```

```
Autocorrelation Function:
Correlation Length 2668000.0
Number of Effective Points: 2.2488755622188905
f(x, n_corr): 1.0004080394534873
```

The correlation length predicted by dividing the chain into many subchains seems reasonable, while that predicted by the autocorrelation function appears to be too large. This could be due to the fact that we should consider variances in model parameters rather than variances in the log likelihood itself. For this MCMC analysis, our model parameters consist of unbiased bin counts of alanine dipeptide configurations having a particular value of ϕ in our discretized coordinates. We could compute the variance of every bin count over the chain, but it is of greater physical interest to consider the distribution of local minima in our unbiased FES over our Metropolis Hastings samples, as these correspond to stable polypeptide conformations. We consider both these measurements of uncertainty in our free energy surface below.

3.1.1 Uncertainty in Free Energy Landscape

```
[15]: # Plot MAP estimate of F with error bars computed from uncertainties in burned␣
 ↪chain
# betaF_MH : rows = MH samples, cols = unbiased bin values of F
# discard burn-in
betaF_MH_burned = betaF_MH[n_burn_idx:][:]

# Compute standard deviations over burned bin counts
betaF_MH_uncert = np.std(betaF_MH_burned, axis=0)
betaF_MH_uncert

plt.errorbar(binC[0], betaF_MAP, yerr=betaF_MH_uncert,
             ecolor="red", barsabove=True, capsize=2.5,
             elinewidth=1, label=r"$F^{ML} \pm \sigma_F^{MH}$")
plt.xlabel(r"Dihedral Angle, $\phi$, $(^\circ)$")
plt.ylabel(r"Free Energy Surface, $\beta F(\phi)$")
plt.xticks(np.arange(-180, 181, 60))
plt.legend(bbox_to_anchor=(1.05, 1.0), loc="upper left")
plt.show()
```



3.1.2 Uncertainty in Minimum Free Energy Diheadrals

We now determine the minima of the maximum likelihood predicted free energy surface and the uncertainty in these parameters from our Metropolis-Hastings sampling of the distribution of these surfaces.

[16]:
```
# Find minima in MAP=ML estimate of FES
minima_MAP_idx = argrelextrema(betaF_MAP, np.less)
stable_phi_MAP = binC[0][minima_MAP_idx[0]]
print("MAP FES:")
print("Minimum FES Phi Angles (degrees):", stable_phi_MAP)
print("")
# Repeat for burned MH estimates -> uncertainty in stable phi angles
n_MH_burned = betaF_MH_burned.shape[0]
stable_phi_MH = np.zeros((n_MH_burned, len(stable_phi_MAP)))
for i in range(n_MH_burned):
    minima_idx = argrelextrema(betaF_MH_burned[i, :], np.less)
    if minima_idx[0].shape[0] > 3:
        betaF_MH_vals = betaF_MH_burned[i, minima_idx[0]]
        phi_vals = binC[0][minima_idx[0]]
        if i % 1000 == 0:
            print("MH Step:", (i+1)*SAVE_MOD_MH)
            print("Min Idx:", minima_idx[0])
            print("betaF:", betaF_MH_vals)
            print("Phi:", phi_vals)
            print("")
```

```
MAP FES:
Minimum FES Phi Angles (degrees): [-147.5  -82.5   82.5]

MH Step: 1000.0
Min Idx: [ 5  7 19 52]
betaF: [-6.75592  -6.81136  -7.4866    0.502372]
Phi: [-152.5 -142.5  -82.5   82.5]

MH Step: 1001000.0
Min Idx: [ 6 19 52 61]
betaF: [-6.95974  -7.45475   0.590543  6.40093 ]
Phi: [-147.5  -82.5   82.5  127.5]

MH Step: 4001000.0
Min Idx: [ 7 18 37 52]
betaF: [-6.88891  -7.39216   9.44655   0.657663]
Phi: [-142.5  -87.5    7.5   82.5]

MH Step: 5001000.0
Min Idx: [ 6 19 37 52]
betaF: [-7.02705  -7.28106   9.39675   0.706194]
Phi: [-147.5  -82.5    7.5   82.5]

MH Step: 6001000.0
Min Idx: [ 7 12 19 37 53]
betaF: [-6.947    -5.31867  -7.28486   9.37184   0.514211]
Phi: [-142.5 -117.5  -82.5    7.5   87.5]
```

We see that `scipy.signal.argrelextrema` is returning more than three stable dihedral ϕ angles for some of the Metropolis-Hastings sampled free energy landscapes. This does not seem reasonable given the previously determined free energy landscapes, and so we formulate our own minimization routine by searching over initial guess ranges of ϕ centered around the expected stable angles.

```
[17]: def find_betaF_min(phi, betaF, ig=[-147.5, -82.5, 82.5], search_range=[30, 30,␣
       ↪30]):
          # Check input
          if len(ig) != len(search_range):
              raise ValueError("Initial Phi Angle Guess and Search Ranges Must have␣
       ↪same length")
```

```
    min_angles = []
    for i in range(len(ig)):
        guess_idx = np.where(phi == ig[i])[0][0]
        upper_idx = np.abs(phi - (ig[i] + search_range[i]/2)).argmin()
        lower_idx = np.abs(phi - (ig[i] - search_range[i]/2)).argmin()
        betaF_search = betaF[lower_idx: upper_idx]
        min_idx = np.argmin(betaF_search) + lower_idx
        min_angles.append(phi[min_idx])
    return min_angles
print("Consistency Check for MAP Estimate:")
print("Minimum FES Phi Angles (degrees):", find_betaF_min(binC[0], betaF_MAP))
```

```
Consistency Check for MAP Estimate:
Minimum FES Phi Angles (degrees): [-147.5, -82.5, 82.5]
```

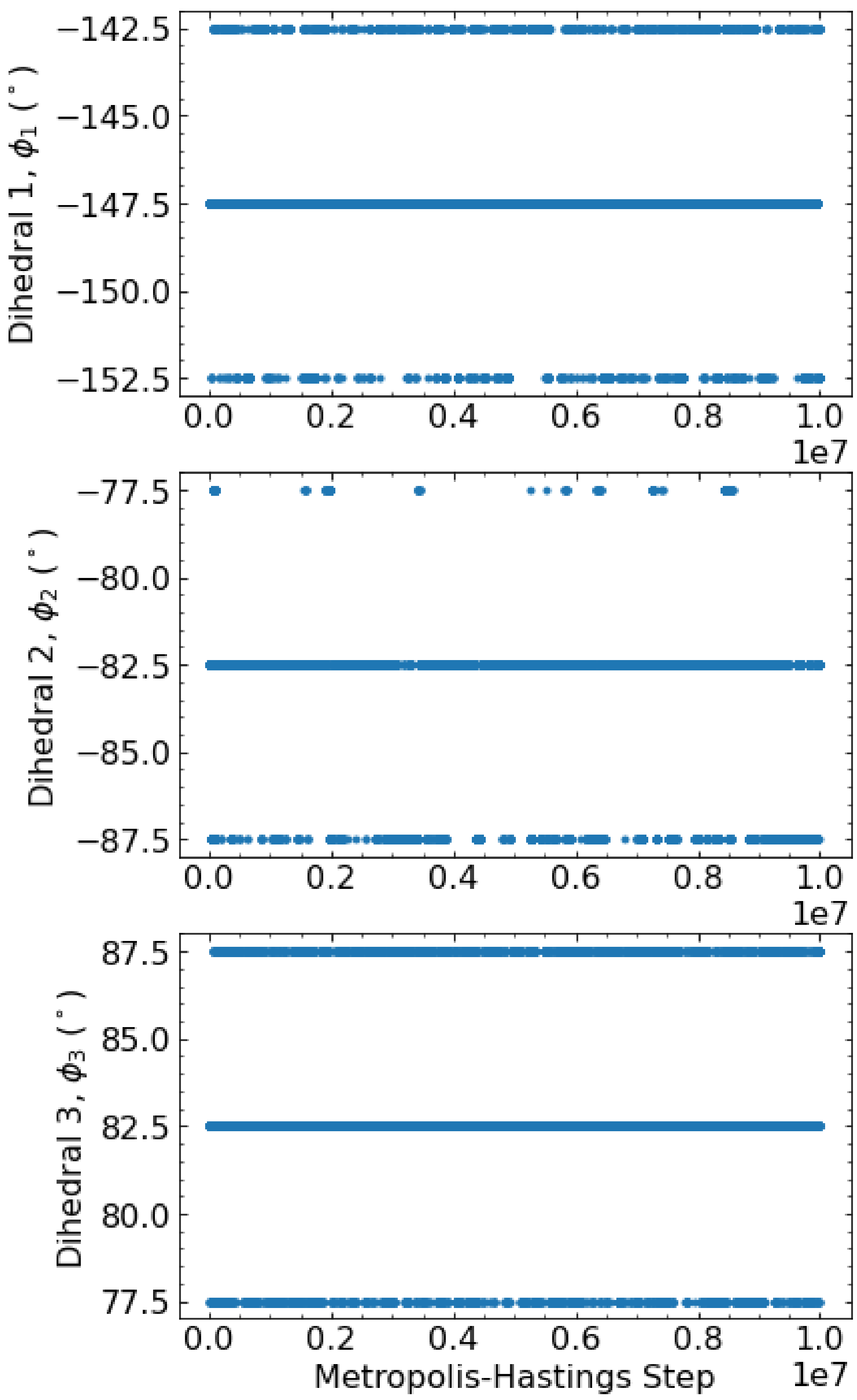
[18]:
```
# Apply find_betaF_min to MH chain
stable_phi_MH_burned = np.zeros((n_MH_burned, len(stable_phi_MAP)))
for i in range(n_MH_burned):
    stable_phi_MH_burned[i, :] = find_betaF_min(binC[0], betaF_MH_burned[i, :])

# Find standard deviation in minimum energy phi dihedrals
uncert_phi = np.std(stable_phi_MH_burned, axis=0)

# Report MAP +/- sigma_MH Estimate
print("Stable Dihedral Phi Angles:")
print("(Maximum Likelihood Estimate) +/- (Standard Deviation Metropolis-Hastings␣
 ↪Samples)")
for i in range(len(stable_phi_MAP)):
    print("(%s +/- %s) degrees" % (stable_phi_MAP[i], uncert_phi[i]))
```

```
Stable Dihedral Phi Angles:
(Maximum Likelihood Estimate) +/- (Standard Deviation Metropolis-Hastings
Samples)
(-147.5 +/- 2.567664733333119) degrees
(-82.5 +/- 2.2137683800225245) degrees
(82.5 +/- 3.0174077538169732) degrees
```

[19]:
```
# Plot progression of minimum phi angles over all MH samples
stable_phi_MH = np.zeros((len(step_MH), 3))
for i in range(len(step_MH)):
    stable_phi_MH[i, :] = find_betaF_min(binC[0], betaF_MH[i, :])

f, axes = plt.subplots(3, figsize=(6, 12))
for i in range(stable_phi_MH.shape[1]):
    axes[i].plot(step_MH, stable_phi_MH[:, i], '.')
    axes[i].set_ylabel(r"Dihedral {x}, $\phi_{x}$ $(^\circ)$".format(x=str(i+1)))
axes[2].set_xlabel(r"Metropolis-Hastings Step")
f.suptitle("Minimum FES Dihedral Angles From M-H Sampling")
```

```
plt.show()
```

Minimum FES Dihedral Angles From M-H Sampling

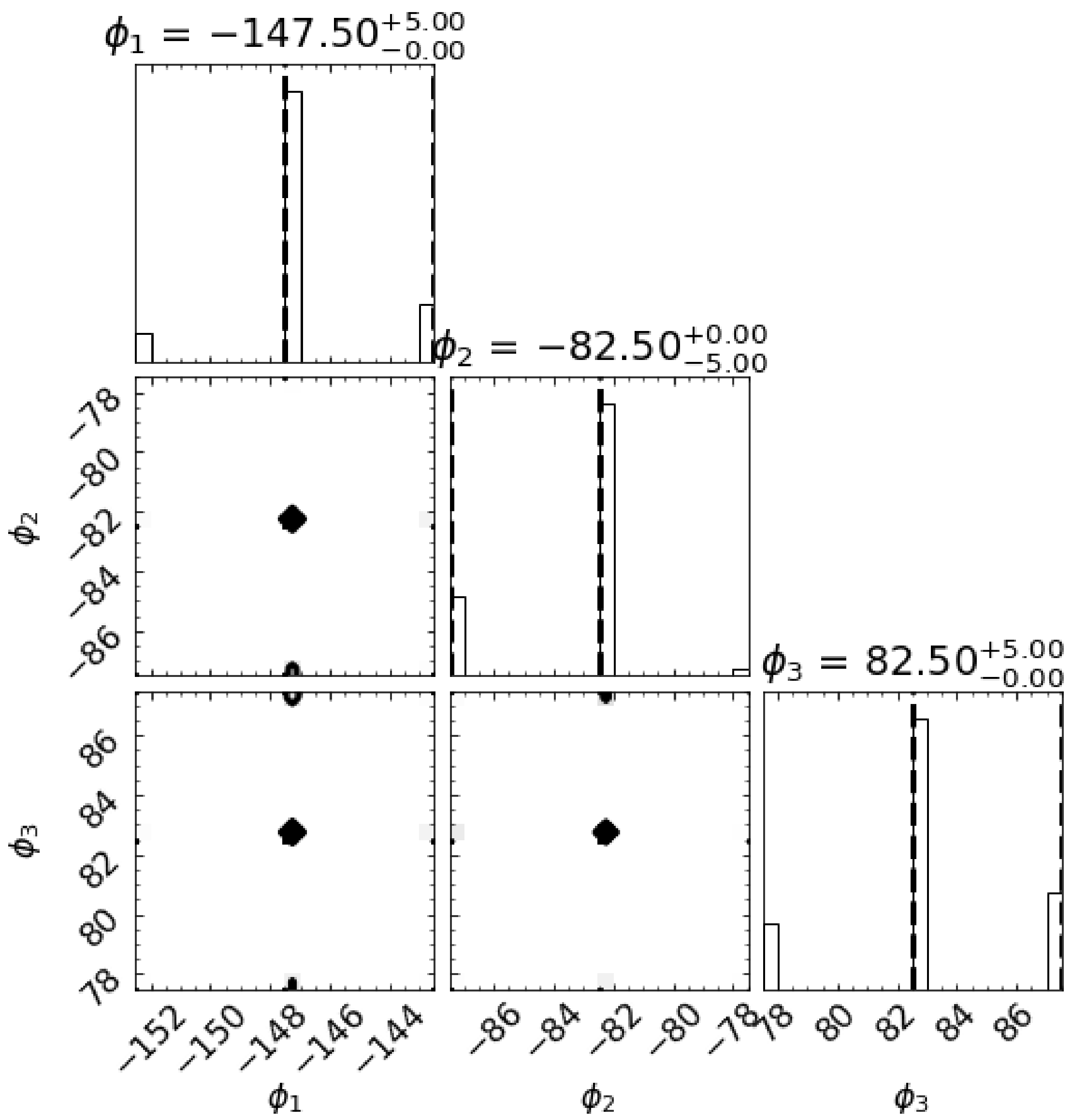


We have thus found that over the course of the MCMC sampling, each stable dihedral ϕ angle determined from the local minima of a posterior distribution-realized free energy surface oscillates between three values, each separated by $5°$ from the maximum likelihood FES predicted angle. This is simply the bin spacing of our order paramter ϕ, indicating that over the course of the random walk the FES shifts the equilibrium dihedral angles by at most plus or minus one bin. Ferguson highlights in his work that "estimations of the unbiased FES are generally relatively robust over a range of bin choices" (Ferguson 1587). The bin size does however introduce bias due to the fact that fewer bins decreases our model complexity (fewer bins means there are less coordinates in our distribution), which should increase the variance of our results due to the bias-variance tradeoff. It would therefore be worthwhile to investigate the effect of varying bin size on these results.

Given the predicted stable dihedral angles over the Metropolis-Hastings steps, we can still formulate corner plots to visualize the covariance between the local minima of the varying FES. We do so using the python `corner` package.

```
[20]: labels = [r"$\phi_1$", r"$\phi_2$", r"$\phi_3$"]
      figure = corner.corner(stable_phi_MH,
                             show_titles=True,
                             labels=labels,
                             plot_datapoints=True,
                             quantiles=[0.16, 0.5, 0.84])
```

```
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
```



The above corner plot thus tells a similar story to the individual time series plots of the stable dihedrals. We again see that each angle determined from the sampled FES jumps between three values, each one ϕ bin apart. The covariance shown between these angles thus shows that that over the course of the random walk, our posterior-realized FES seems to simply translate as a whole while maintaining its general shape.

3.1.3 Improving Uncertainty Quantification Using Parallel Tempering

We now attempt to improve the uncertainty in the FES landscape determined by Metropolis-Hastings sampling by instead using the method of parallel tempering. We therefore must first quantify the overall uncertainty in the FES landscape determined by MCMC. We currently have an estimate of the FES at each discrete location of ϕ determined by maximization of the posterior distribution (in this case maximizatin of the likelihood due to the choice of a uniform prior), along with an estimate of the uncertainty in the free energy at each ϕ coordinate given by the standard deviation in the free energy at this coordinate over the course of our MCMC samples. We therefore quantify the uncertainty in the landscape as a whole by taking the RMSD over our discrete ϕ bins. This is equivalent to the square root of the average squared deviation in each discrete free energy value over the course of the MCMC random walk, i.e:

$$\text{RMSD} = \sqrt{\frac{\sum_i (x_i - \hat{x}_i)^2}{N}} = \sqrt{\frac{\sum_i (\sigma_i^{MH})^2}{N}}$$

where the sum is over our discrete ϕ points, $\hat{x}_i$ is the MAP-predicted free energy, $x_i = \hat{x}_i + \sigma_i^{MH}$, and N is the total number of points in a landscape.

Computing this quantity from the Metropolis-Hastings random walk, we find:

[21]:
```python
# Find RMSD for the MH data
# uncertainties contained in betaF_MH_uncert
rmsd_MH = np.sqrt( np.sum(betaF_MH_uncert**2)/len(betaF_MH_uncert) )
print("RMSD in beta*F from M-H Sampling (Dimensionless):", rmsd_MH)
```

```
RMSD in beta*F from M-H Sampling (Dimensionless): 0.09145229219398845
```

We now attempt to improve this metric using parallel tempering. The procedure of parallel tempering (also known as replica exchange in the context of MD simulations) is a technique that allows for greater exploration of the posterior distibution over the course of the MCMC random walk. It does so by running multiple MCMC chains in parallel (and therefore is highly applicable to parallel computation), and within each we optimize our likelihood scaled by an additional "temperature" term T, resulting in a "tempered" distribution. In traditional Monte Carlo simulations of molecular systems, this would correspond to the actual temperature of the ensemble for a given chain, but here it simply acts as a model hyperparameter. One then propagates each chain according to the random walk of a traditional MCMC procedure, but we now allow each chain to exchange temperatures with one of its neighbors at each proposal step and include this transition in our acceptence criteria. This allows for a greater exploration of parameter space as chains have an even greater ability to overcome potential energy barriers that would otherwise trap them in local minima.

This procedure has been extended to apply to any MCMC sampling procedure. Following "A Parallel Tempering algorithm for probabilistic sampling and multimodal optimization" by Malcolm Sambridge, we define our objective function to be optimized as:

$$\pi(\underline{m}, T) = \exp\left(-\phi(\underline{m})/T\right)$$

Here ϕ is unrelated to the dihedrals under consideration but instead represents a general function that we wish to optimize without any tempering. π is then a probability distribution taking the role of our likelihood, and it is thus clear from the expression above that ϕ takes the role of χ^2, i.e:

$$\phi \to \chi^2$$
$$\pi = e^{-\phi/T} \to e^{-\chi^2/T} \to \exp\left[(\ln \mathcal{L})/T\right]$$

Sambridge then show that this modification leads to the following updated acceptence criterion for moving from state *i* to state *j* in our random walk:

$$\alpha(i,j) = \min\left\{1, \exp\left[\left(\frac{1}{T_i} - \frac{1}{T_j}\right)\left(\phi(\underline{m}_i) - \phi(\underline{m}_j)\right)\right]\right\}$$

Sambridge also provides pseudo-code for this parallel tempering exchange procedure in Algorithm 1 of his work. We summarize these steps below:

```
def BayesPT(temps, nMCMC):
    nChains = len(temps)
    # Loop over time step of MC chains
    for j in range(nMCMC):
        # Loop over temperatures
        # also defines the number of chains
        for i in range(nChains):
            # Advance chain i over step j
            # according to Ferguson's Algorithm 2
            # Including the temp in the objective func
            pi_ij = AdvanceChain(i, j, Ti)
        # Now swap temperatures of random pairs of chains
        for i in range(nChains):
            p = randint(nChains)  # choose random chain
            q = randint(nChains) != p  # choose a random partner to swap temp with
            r1 = [pi_qj/pi_pj]^(1/Tp)
            r2 = [pi_pj/pi_qj]^(1/Tq)
            alpha = min(1, r1r2)
            if alpha < np.rand(0, 1):
                # swap temps
                temp = Tp
                Tp = Tq
                Tq = temp
```

We again note that our tempered distribution π is related to the likelihood $\mathcal{L}$ by:

$$\pi = e^{(-\phi/T)} \sim e^{(-\chi^2/T)} \sim e^{(-1/T)(-\ln\mathcal{L})} \sim e^{\ln\mathcal{L}^{(1/T)}} \sim \mathcal{L}^{(1/T)}$$

We then see that the above algorithm involves ratios of π for a current and proposed step of a given chain. It will therefore be helpful for numerics to instead work with logarithms of π so that we can in turn work with log likelihoods. These are related as follows:

$$\pi = \mathcal{L}^{(1/T)} \implies \ln\pi = \frac{1}{T}\ln\mathcal{L}$$

$$\frac{\pi_{\text{new}}}{\pi_{\text{old}}} = \pi_{\text{new}} e^{\ln\frac{1}{\pi_{\text{old}}}} = e^{\ln\pi_{\text{new}}} e^{-\ln\pi_{\text{old}}} = \exp\left[\ln\pi_{\text{new}} - \ln\pi_{\text{old}}\right]$$

$$\frac{\pi_{\text{new}}}{\pi_{\text{old}}} = \exp\left[\frac{1}{T}\ln\mathcal{L}_{\text{new}} - \frac{1}{T}\ln\mathcal{L}_{\text{old}}\right]$$

Where T is the temperature of the current chain that we're propagating. We perform this calculation for two chains, each at a different temperature, and swap their temperatures before computing the acceptence criterion.

We thus implement the algorithm above, recording the propagation of each of our chains. We can then quantify the efffectiveness of parallel tempering as compared with traditional MCMC by averaging the log likelihood over all of our chains and considering this average value's propagation over the MCMC steps. Further, we can compute a free energy surface for each chain at each step of the random walk and determine the RMSD over this larger collection of distributions.

We begin by formulating a parallel tempering MCMC procedure for the alanine dipeptide data set by combining Algorithm 2 of Ferguson's work with Algorithm 1 of Sambridge's work. We now require a three dimensional data structure to store the results of each chain's random walk. Visualizing this structure as a set of square sheets extending into/out of the page, each sheet represents a different Markov Chain (propagated at a different temperature) with the rows and columns of each sheet corresponding to MCMC steps and unbiased bin counts as Ferguson previously developed. We again consider the simpler case of a uniform prior here so that our objective function is simply the likelihood defined in Ferguson's work, repeated below:

$$\mathcal{L} = P(\{n_{i,l}\}|\{p_l\}) = D(\{n_{i,l}\},\{c_{i,l}\})\prod_{l=1}^{M} p_l^{M_l}\prod_{i=1}^{S}\left(\sum_{l=1}^{M} p_l c_{i,l}\right)^{-N_i}$$

We can then drop the constant $D(\{n_{i,l}\},\{c_{i,l}\})$ that is independent of our model unbiased distribution $\{p_l\}$, since this will cancel when we apply the acceptence criteria. Dropping this constant and takng the logarithm, we have:

$$\ln\mathcal{L} = \ln\left[\prod_{l=1}^{M} p_l^{M_l}\prod_{i=1}^{S}\left(\sum_{l=1}^{M} p_l c_{i,l}\right)^{-N_i}\right]$$

$$\ln\mathcal{L} = \ln\left[\prod_{l=1}^{M} p_l^{M_l}\right] + \ln\left[\prod_{i=1}^{S}\left(\sum_{l=1}^{M} p_l c_{i,l}\right)^{-N_i}\right]$$

$$\ln\mathcal{L} = \sum_{l=1}^{M}\left(\ln\left(p_l^{M_l}\right)\right) + \sum_{i=1}^{S}\left(\ln\left(\sum_{l=1}^{M} p_l c_{i,l}\right)^{-N_i}\right)$$

$$\ln\mathcal{L} = \sum_{l=1}^{M} M_l \ln p_l - \sum_{i=1}^{S} N_i \ln\left(\sum_{l=1}^{M} p_l c_{i,l}\right)$$

Finally, we note that the argument of the second logarithm in the last expression, $\sum_{l=1}^{M} p_l c_{il}$, can be thought of as a matrix of bias boltzmann factor c_{il} whose rows correspond to simulations and columns the bins of our unbiased distribution multiplying a column vector representing the distribution itself.

[49]:
```
# Code in this cell taken directly from Ferguson's `BayesWHAM.py`
import math


kB = 0.0083144621  # Boltzmann's constant / kJ/mol.K
T = 298  # temperature in (K)
beta = 1/(kB*T)
# Load biased simulation bin counts
n_il = []
S = 18
M = n_bins
histDir = "diala_phi_EXAMPLE/hist"

for i in range(0,S):
    hist_filename = histDir + '/hist_' + str(i+1) + '.txt'
    hist_DATA= []
    with open(hist_filename,'r') as fin:
        for line in fin:
            hist_DATA.append(line.strip().split())
    if len(hist_DATA) != 1:
        print("\nERROR - Did not find expected row vector in reading %s" %␣
 ↪(hist_filename))
        sys.exit(-1)
    if len(hist_DATA[0]) != M:
        print("\nERROR - Row vector in %s did not contain expected number of␣
 ↪elements M = M_1*M_2*...*M_dim = %d given histogram bins specified in %s" %␣
 ↪(hist_filename,M,histBinEdgesFile))
        sys.exit(-1)
    n_il.append(hist_DATA[0])
n_il = [[float(y) for y in x] for x in n_il]
n_il = np.array(n_il)

# precomputing aggregated statistics
N_i = np.sum(n_il,axis=1)         # total counts in simulation i
M_l = np.sum(n_il,axis=0)         # total counts in bin l

# Load the bias boltzmann factor matrix
dim = 1
periodicity = "[1]"
periodicity = periodicity[1:-1].split(',')
periodicity = [int(x) for x in periodicity]

def ind2sub_RMO(sz,ind):
    if (ind < 0) | (ind > (np.prod(sz)-1)):
        print("\nERROR - Variable ind = %d < 0 or ind = %d > # elements in␣
 ↪tensor of size sz = %d in ind2sub_RMO" % (ind,ind,np.prod(sz)))
        sys.exit(-1)
    sub = np.zeros(sz.shape[0], dtype=np.uint64)
    for ii in range(0,len(sz)-1):
        P = np.prod(sz[ii+1:])
        sub_ii = math.floor(float(ind)/float(P))
        sub[ii] = sub_ii
        ind = ind - sub_ii*P
    sub[-1] = ind

    return sub

histBinEdgesFile = os.path.join("diala_phi_EXAMPLE", "hist", "hist_binEdges.txt")
harmonicBiasesFile = os.path.join("diala_phi_EXAMPLE", "bias", "harmonic_biases.
 ↪txt")

binE = []
with open(histBinEdgesFile,'r') as fin:
    for line in fin:
        binE.append(line.strip().split())

binE = [[float(y) for y in x] for x in binE]
binE = np.array(binE)

period = np.zeros(dim)
for d in range(0,dim):
    if periodicity[d] == 0:
        period[d] = float('nan')
    else:
        period[d] = binE[d][-1] - binE[d][0]

M_k = np.zeros(dim, dtype=np.uint64)
for d in range(0,dim):
    M_k[d] = len(binE[d])-1

harmonicBiasesFile_DATA = []
with open(harmonicBiasesFile,'r') as fin:
    for line in fin:
        harmonicBiasesFile_DATA.append(line.strip().split())

umbC = [item[1:1+dim] for item in harmonicBiasesFile_DATA]
umbC = [[float(y) for y in x] for x in umbC]
umbC = np.array(umbC)

umbF = [item[1+dim:1+2*dim] for item in harmonicBiasesFile_DATA]
umbF = [[float(y) for y in x] for x in umbF]
umbF = np.array(umbF)
```

```
c_il = np.zeros((S,M))  # S-by-M matrix containing biases due to artificial␣
↪harmonic potential for simulation i in bin l
for i in range(0,S):
    for l in range(0,M):
        sub = ind2sub_RMO(M_k,l)
        expArg = 0
        for d in range(0,dim):
            if periodicity[d] != 0:
                delta = min( abs(binC[d][sub[d]]-umbC[i,d]),␣
↪abs(binC[d][sub[d]]-umbC[i,d]+period[d]),␣
↪abs(binC[d][sub[d]]-umbC[i,d]-period[d]) )
            else:
                delta = abs(binC[d][sub[d]]-umbC[i,d])
            expArg = expArg + 0.5*umbF[i,d]*math.pow(delta,2)
        c_il[i,l] = math.exp(-beta*expArg)
```

[51]:
```
# Define log likelihood
def loglike(pl, Ml, Ni, cil):
    """
    Returns log likelihood for a given unbiased distribution (model)
    pl  : unbiased distribution (our model), shape 1 x n_bins
    Ml  : Total counts in bin l aggregated over all sims, shape 1 x n_bins
    Ni  : Total counts in sim i aggregated over all bins, shape n_sims x 1
    cil : bias boltzmann factor matrix, rows=sims, cols=bins, shape n_sims x␣
↪n_bins
    """
    sim_log_arg_i = np.dot(cil, np.transpose(pl))
    result = np.sum(Ml*np.log(pl)) - np.sum(Ni*sim_log_arg_i)
    return result
```

[52]:
```
# Check for consistency in the above loglike function using the MAP determined␣
↪distribution p_l
# This was already computed via Ferguson's code
# Results in `diala_phi_EXAMPLE/p_MAP.txt`
f__p_MAP = os.path.join("diala_phi_EXAMPLE", "BayesWHAM_OUTPUT", "p_MAP.txt")
with open(f__p_MAP, "r") as fin:
    line = fin.readline()
    p_MAP = line.split()
p_MAP = np.asarray(p_MAP, dtype="float").reshape((1, -1))
print(p_MAP.shape)
print(M_l.shape)
print(N_i.shape)
print(c_il.shape)
print("Log Likelihood, MAP PDF, Function Defined Above:", loglike(p_MAP, M_l,␣
↪N_i, c_il))
```

```
f__logL_MAP = os.path.join("diala_phi_EXAMPLE", "BayesWHAM_OUTPUT", "logL_MAP.
 ↪txt")
with open(f__logL_MAP, "r") as fin:
    logL_MAP_paper = float(fin.readline().strip())
print("Log Likelihood, MAP PDF, Paper Result:", logL_MAP_paper)
```

```
(1, 72)
(72,)
(18,)
(18, 72)
Log Likelihood, MAP PDF, Function Defined Above: -157884.9373226053
Log Likelihood, MAP PDF, Paper Result: -13493.5
```

It thus appears that something may not be correct in the above definition of the log likelihood, given that using it to calculate the log likelihood for the maximum posterior distribution of unbiased counts which gave the initial βF vs ϕ curve is lower than the paper reported value by a factor of 10. Both results are only up to an additive constant, so some deviation is expected, but the difference shown above is not promising given our goal is to maximimize the log likelihood. We therefore need to revisit this issue, but for now we proceed with constructing the parallel tempering MCMC procedure given the log likelihood defined above.

[22]:
```
# Initialize PTMCMC parameters and data structures
n_visits = 0
N_PTMCMC = int(1e3)  # number of MCMC steps per chain
# Start with 10 chains, will want to play with this:
N_CHAINS = 10
T_MIN = 100
T_MAX = 1000  # more hyperparams to tune
n_bins  = len(binC[0])  # same phi spacing as previously used
temps = np.linspace(T_MIN, T_MAX, N_CHAINS)
# Data structure to store results (biased distribution over phi bins)
ptmcmc_results = np.zeros((N_PTMCMC, n_bins, N_CHAINS))
```

[55]:
```
# Check Normalization of p_MAP
print("Sum of Over Bins of MAP Biased Distribution Estimate:", np.sum(p_MAP))
```

```
Sum of Over Bins of MAP Biased Distribution Estimate: 1.0000000301062002
```

[96]:
```
# Initial guess for unbiased distribution = MAP paper result, set this for each␣
 ↪chains
p_k_T = np.tile(p_MAP, (N_CHAINS, 1))  # rows - chains, cols = bins
dp = 5e-4  # Max step size, tune to achieve ~25% acceptance
log_pi = np.zeros((N_PTMCMC, N_CHAINS))  # log(pi) = (1/T)log(L), each chain at␣
 ↪different T
# Loop over MCMC steps:
for j in range(N_PTMCMC):
    # Loop over chains, each of which is at a different temperature
    for i in range(N_CHAINS):
        # Advance each chain by varying a randomly selected ubiased bin ratio (a␣
↪model param)
        # Compute the new updated likelihood for the current chain's current step
        # Save current state
        p_k_T_prime = p_k_T
        # Select a random bin for each chain to maintain detailed balance
        rand_bin = np.random.randint(M)  # random idx 0...M-1
        # Perform symmetric walk trial move for current chain
        p_k_T[i, rand_bin] = p_k_T[i, rand_bin] + (2*np.random.uniform(0, 1) - 0.
↪5)*dp
        # Renormalize the resulting distribution
        p_k_T[i, :] = p_k_T[i, :]/np.sum(p_k_T[i, :])
        # Find pi
        log_pi[j, i] = (1/temps[i])*loglike(p_k_T[i, :], M_l, N_i, c_il)
```

```
/opt/anaconda3/envs/PythonData/lib/python3.7/site-
packages/ipykernel_launcher.py:11: RuntimeWarning: invalid value encountered in
log
  # This is added back by InteractiveShellApp.init_path()
```

[ ]:

```
```

We leave building a parallel tempering MCMC routine from scratch aside for now, and instead investigate its effectiveness using the package `emcee`. We have already defined our log likelihood function, and in addition we use a flat prior:

[98]:

```
# Define prior
def logp(pl):
    return 0.0
```

[107]:

```
# Construct PT MCMC walkers following the emcee documentation
ntemps = 20
nwalkers = 100
ndim = M

sampler = ptsampler.PTSampler(ntemps, nwalkers, ndim, loglike, logp)
```

```
        ---------------------------------------------------------------------------

        ImportError                               Traceback (most recent call last)

        <ipython-input-107-411f20d5474d> in <module>
          4 ndim = M
          5
    ----> 6 sampler = ptsampler.PTSampler(ntemps, nwalkers, ndim, loglike, logp)
        /opt/anaconda3/envs/PythonData/lib/python3.7/site-packages/emcee/ptsampler.
    ↪py in __init__(self, *args, **kwargs)
          9         def __init__(self, *args, **kwargs):
         10             raise ImportError(
    ---> 11                 "The PTSampler from emcee has been forked to "
         12                 "https://github.com/willvousden/ptemcee, "
         13                 "please install that package to continue using the␣
    ↪PTSampler"


        ImportError: The PTSampler from emcee has been forked to https://github.
    ↪com/willvousden/ptemcee, please install that package to continue using the␣
    ↪PTSampler
```

```
[ ]:
```

4 References

Ferguson, A.L., "BayesWHAM: A Bayesian approach for free energy estimation, reweighting, and uncertainty quantification in the weighted histogram analysis method" J. Comput. Chem. 38 18 1583-1605 (2017). DOI: 10.1002/jcc.24800

Mironov, V., Alexeev, Y., Mulligan, V. K., Fedorov, D. G., "A systematic study of minima in alanine dipeptide" J. Comput. Chem. 40 297–309 (2019). DOI: 10.1002/jcc.25589

Sambridge, M., "A Parallel Tempering algorithm for probabilistic sampling and multimodal optimization", Geophysical Journal International, 196 1 357-374 (2014). DOI: https://doi.org/10.1093/gji/ggt342