0.0.1 Part 1. Define the Log Likelihood function

Given a set of data X and a model $M(\theta)$ on this data with parameters θ, we can define the likelihood function as the probability that the model with parameters θ describes the data:

$$L(\theta; X) = P(X|\theta)$$

Usually, we work in the log space as the log-likelihood is easier to define and optimize. As the log function is monotonic, anything that yields the maximum or minimum log-likelihood would then also yield the maximum or minimum likelihood. The log-likelihood is defined as

$$\ell(\theta; X) = \log(L(\theta; X))$$

Using this chi square given above, define the priors and the log likelihood function.

```
In [194]: names = ['logMmin', 'sigma_logM', 'logM0', 'logM1', 'alpha',
                   'mean_occupation_centrals_assembias_param1',
                   'mean_occupation_satellites_assembias_param1']
          bounds = [(9.0, 14.0), (0.01, 1.5), (9.0, 14.0), (10.7, 15.0), (0.0, 2.0), (-1.0, 1.0), (-1.0

          def log_likelihood(theta):
              theta = theta.copy()
              model.param_dict.update(dict(zip(names, theta)))
              n_mod, wp_mod = halotab.predict(model)
              return -0.5 * (np.dot(wp_mod - wp_obs, np.dot(np.linalg.inv(wp_cov), (wp_mod - wp_obs))) 
```

```
In [195]: grader.check("q1.1")
```

```
Out[195]: q1.1 results: All test cases passed!
```

0.0.2 Part 2. Maximum Likelihood Estimate / Maximum A Posteriori

Given a log likelihood function $\ell(\theta; X)$, in the frequentist set-up we can calculate the maximum likelihood estimate (MLE) of our parameters θ by finding the MLE best fit parameters $\hat{\theta}$

$$\hat{\theta} = \text{argmax}_{\theta}\ \ell(\theta; X)$$

However, in this problem, we don't just want to find the *best fit parameters* $\hat{\theta}$, but we also want to find the *uncertainties* of these best fit parameters, as well as the covariances of these parameters and their distribution in parameter space.

The Fisher Information Matrix could help us find the parameter covariance. Recall that the Fisher Matrix is defined as

$$F_{ij} = -\left\langle \frac{\partial^2}{\partial\theta_i \partial\theta_j} \ell(\theta; X) \right\rangle_{\theta_{MLE}}$$

(Note that as described in the lecture notes, the fisher matrix is evaluated at the best-fit MLE parameters) The Fisher matrix also has the property such that its inverse yields the covariance matrix.

$$C^{-1} = F$$

To compute the Fisher information, we make use of the Hessian matrix of the log-likelihood. This is defined as

$$H_{ij} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} \ell(\theta; X)$$

which means that the Fisher Information matrix is the negative expectation of the Hessian

$$F = -\langle H \rangle$$

Note: this is only true in the assumption of large, independent samples from the likelihood, when distribution on θ exhibits asymptotic normality.

$$\hat{\theta} \sim N\left(\theta, \frac{F(\theta)^{-1}}{n}\right)$$

By the Laplace approximation, this indicates that the Hessian would converge to the Fisher Information Matrix.

In practice, this symbolizes that the Hessian of the log likelihood can be used to approximate the Fisher matrix. Under the assumption that the parameter space is Gaussian distributed then, we can use the Hessian of the likelihood to calculate the covariance matrix and plot Gaussian stair plots showing the distribution in parameter space.

The goal of this section is to accomplish this task. Find the parameters that maximize the log likelihood within the prior bounds given above, and plot the parameter distributions for the parameters given, using the prior bounds as bounds of each of the stair plots. To help in the plotting part, a skeleton stair plot code is provided, but you will have to modify the code to fit the parameters. (Note that technically, we are conducting Maximum A Posteriori (MAP) estimation of parameters as we're incorporating the uniform priors given earlier)

Hints:

1. `scipy.optimize` allows the use of optimization methods with bounds on optimization parameters. Read the documentation and use a method that is able to do bounded optimization on the likelihood.
2. To compute the Hessian on the log-likelihood function, use numerical finite difference methods and not automatic differentiation methods. This is because the likelihood function you constructed in Part 1 utilises packages which are not automatically differentiable by autograd or pytorch. The package `numdifftools` contains the function `numdifftools.Hessian` which will help you in this area.

```
In [196]: labels = [r'$\log M_{min}$', r'$\sigma_{\log M}$',r'$\log M_0$',r'$\log M_1$',r'$\alpha$',r'$

          from scipy.optimize import minimize

          def neg_log_likelihood(theta):
              return -log_likelihood(theta)

          x0 = np.array([11.5, 0.755, 11.5, 12.85, 1.0, 0.0, 0.0])
          opt_p = minimize(neg_log_likelihood, x0, method='L-BFGS-B', bounds=bounds)

          print(opt_p.x)
          for i in range(len(labels)):
              print(r"MLE value of %s = %.5f" %(labels[i], opt_p.x[i]))
```

```
[12.15663725  0.66780333 12.18372699 13.26765914  1.06058745  1.
 -0.06105078]
MLE value of $\log M_{min}$ = 12.15664
MLE value of $\sigma_{\log M}$ = 0.66780
MLE value of $\log M_0$ = 12.18373
MLE value of $\log M_1$ = 13.26766
MLE value of $\alpha$ = 1.06059
MLE value of $A_c$ = 1.00000
MLE value of $A_s$ = -0.06105
```

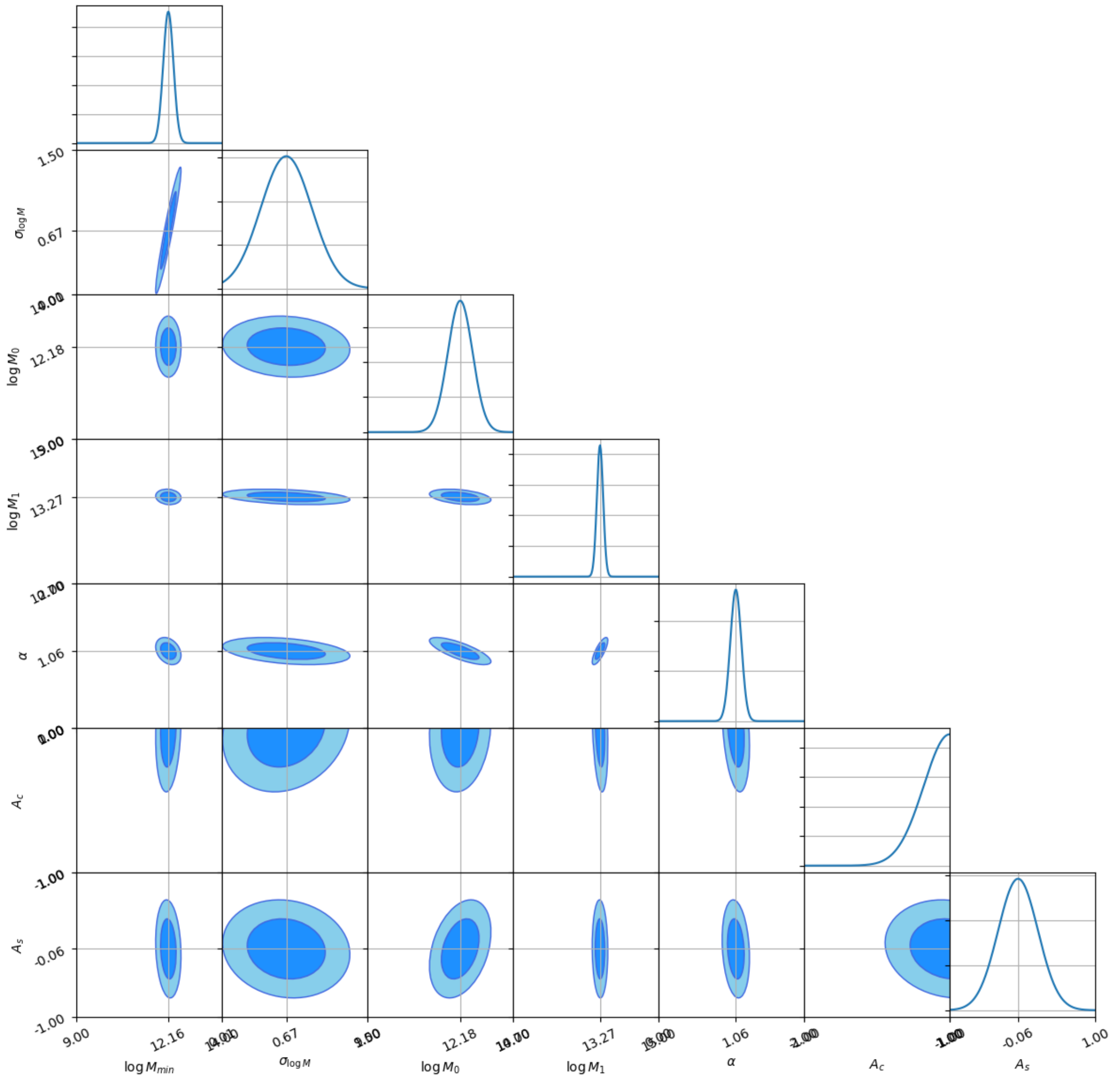
```
In [197]: hess_func = numdifftools.Hessian(log_likelihood)
          hessian = hess_func(opt_p.x)
          fisher = -hessian
          cov_matrix = np.linalg.inv(fisher)

          print("Covariance matrix computed successfully")
          print("Parameter uncertainties (1-sigma):")
          for i in range(len(labels)):
              print("%s: %.5f +/- %.5f" % (labels[i], opt_p.x[i], np.sqrt(cov_matrix[i, i])))
```

```
Covariance matrix computed successfully
```

```
Parameter uncertainties (1-sigma):
$\log M_{min}$: 12.15664 +/- 0.17555
$\sigma_{\log M}$: 0.66780 +/- 0.26266
$\log M_0$: 12.18373 +/- 0.42571
$\log M_1$: 13.26766 +/- 0.09287
$\alpha$: 1.06059 +/- 0.07573
$A_c$: 1.00000 +/- 0.35809
$A_s$: -0.06105 +/- 0.27369
```

```
In [198]: from scipy.stats import norm

          fig, axes = plt.subplots(7, 7, figsize=(14, 14))
          fig.subplots_adjust(wspace=0, hspace=0)
          p_tex = labels

          for i in range(7):
              for j in range(7):
                  ax = axes[i, j]
                  if j > i:
                      ax.axis('off')
                      continue
                  elif i == j:
                      ax.grid(True)
                      xarr = np.linspace(bounds[i][0], bounds[i][1], 200)
                      yarr = norm.pdf(xarr, loc=opt_p.x[i], scale=np.sqrt(cov_matrix[i, i]))
                      ax.plot(xarr, yarr)
                      ax.set_xlim(bounds[i][0], bounds[i][1])
                      ax.set_xticks([bounds[i][0], opt_p.x[i], bounds[i][1]])
                      ax.set_yticklabels([])
                      ax.set_xticklabels([])
                  else:
                      ax.grid(True)
                      CovM = cov_matrix[np.ix_([j, i], [j, i])]
                      eigvec, eigval, u = np.linalg.svd(CovM)
                      semimaj = np.sqrt(eigval[0]) * 2.
                      semimin = np.sqrt(eigval[1]) * 2.
                      theta = np.arctan(eigvec[0][1] / eigvec[0][0])
                      ell = mpl.patches.Ellipse(xy=[opt_p.x[j], opt_p.x[i]], width=1.52*semimaj, height
                                                angle=theta*180/np.pi, facecolor='dodgerblue', edgecolor
                                                label='68% confidence')
                      ell2 = mpl.patches.Ellipse(xy=[opt_p.x[j], opt_p.x[i]], width=2.48*semimaj, heigh
                                                 angle=theta*180/np.pi, facecolor='skyblue', edgecolor='
                                                 label='95% confidence')
                      ax.add_patch(ell2)
                      ax.add_patch(ell)
                      ax.set_ylim(bounds[i][0], bounds[i][1])
                      ax.set_xlim(bounds[j][0], bounds[j][1])
                      ax.set_xticks([bounds[j][0], opt_p.x[j], bounds[j][1]])
                      ax.set_yticks([bounds[i][0], opt_p.x[i], bounds[i][1]])

                  if j != 0:
                      ax.set_yticklabels([])
```

```
        if i != 6:
            ax.set_xticklabels([])
        if j == 0 and i != 0:
            ax.set_ylabel(p_tex[i], fontsize=10)
            ax.set_yticklabels([f'{bounds[i][0]:.2f}', f'{opt_p.x[i]:.2f}', f'{bounds[i][1]:.
            [tl.set_rotation(26) for tl in ax.get_yticklabels()]
        if i == 6:
            ax.set_xlabel(p_tex[j], fontsize=10)
            ax.set_xticklabels([f'{bounds[j][0]:.2f}', f'{opt_p.x[j]:.2f}', f'{bounds[j][1]:.
            [tl.set_rotation(26) for tl in ax.get_xticklabels()]

plt.show()
```



```
In [199]: grader.check("q1.2")
```

```
Out[199]: q1.2 results: All test cases passed!
```

0.0.3 Part 3. Markov Chain Monte Carlo

In many cases, the likelihood or the posterior distribution over parameter space is highly non-Gaussian. When the Gaussian approximation is no longer sufficient for parameter distribution estimation and sampling, we can instead use Markov Chain Monte Carlo methods to do sampling to get a better estimate for the distribution of parameters.

Here, we will use the `pocomc` package as an example case for parameter sampling estimation. Following the example, you are expected to conduct the same type of sampling with your own Metropolis-Hastings algorithm, and get the same resulting plot.

So we define the priors as a list to the `pc.Prior` function, each item in the list should be a uniform random variable given by `uniform(lower, upper-lower)`.

```
In [200]: import pocomc as pc
```

```
In [201]: prior = pc.Prior([
              uniform(9,14-9),
              uniform(0.01,1.5-0.01),
              uniform(9,14-9),
              uniform(10.7,15.0-10.7),
              uniform(0,2-0),
              uniform(-1,+1-(-1)),
              uniform(-1,+1-(-1)),
          ])
```

```
In [202]: sampler = pc.Sampler(prior=prior, likelihood=log_likelihood)
          sampler.run()
          samples, weights, logl, logp = sampler.posterior()
          logz, logz_err = sampler.evidence()
          print('logZ = ', np.round(logz,4), '+-', np.round(logz_err,4))
```

```
Iter: 36it [01:23,  2.31s/it, beta=1, calls=46592, ESS=3985, logZ=-24.8, logP=-18, acc=0.638, steps=7,
```

```
logZ =  -24.9005 +- 0.0377
```

```
In [203]: corner.corner(samples, weights=weights, color='C0', smooth=1.0,
                            labels=['logMmin', 'sigma_logM', 'logM0', 'logM1', 'alpha', 'A_c', 'A_s'],
                            range=[(9,14),(0.01,1.5),(9,14),(10.7,15.0),(0,2),(-1,+1),(-1,+1)]);
```

```
In [204]: mcmc_means = np.average(samples, weights=weights, axis=0)
          mcmc_stds = np.sqrt(np.average((samples - mcmc_means)**2, weights=weights, axis=0))

          for i in range(len(labels)):
              print("%s = %.5f +/- %.5f" % (labels[i], mcmc_means[i], mcmc_stds[i]))
```

```
$\log M_{min}$ = 12.23592 +/- 0.17564
$\sigma_{\log M}$ = 0.73282 +/- 0.23767
$\log M_0$ = 11.16319 +/- 0.92317
$\log M_1$ = 13.31986 +/- 0.08199
$\alpha$ = 1.10293 +/- 0.05350
$A_c$ = 0.78939 +/- 0.18610
$A_s$ = -0.02259 +/- 0.24595
```

```
In [205]: for i in range(len(labels)):
              sigma_away = np.abs(opt_p.x[i] - mcmc_means[i]) / mcmc_stds[i]
              print("%s : MLE values %.2f sigma away from MCMC mean" % (labels[i], sigma_away))
```

```
$\log M_{min}$ : MLE values 0.45 sigma away from MCMC mean
$\sigma_{\log M}$ : MLE values 0.27 sigma away from MCMC mean
$\log M_0$ : MLE values 1.11 sigma away from MCMC mean
$\log M_1$ : MLE values 0.64 sigma away from MCMC mean
$\alpha$ : MLE values 0.79 sigma away from MCMC mean
$A_c$ : MLE values 1.13 sigma away from MCMC mean
$A_s$ : MLE values 0.16 sigma away from MCMC mean
```

The results from this corner plot should look similar to your results from Part 2 using MLE. (If they don't, double check your log-likelihood function or your MLE code). We see however that the results are not quite Gaussian, especially given our priors. Our goal then is to reproduce this work using a simpler Markov Chain Monte Carlo algorithm: Metropolis Hastings.

Write your own MCMC algorithm using Metropolis Hastings, and sample from the posterior distribution and make a corner plot like the one above. Report the mean and standard deviations of your sampled parameters, and plot the convergences of your chains as a function of timestep. Lastly, discuss the differences your simple MCMC alrogithm have with the pocomc plot above, the MLE plot from Part 2 as well as Figure 6 from https://arxiv.org/pdf/1606.07817. Remember to leave comments in your code and explain your steps.

Hints:

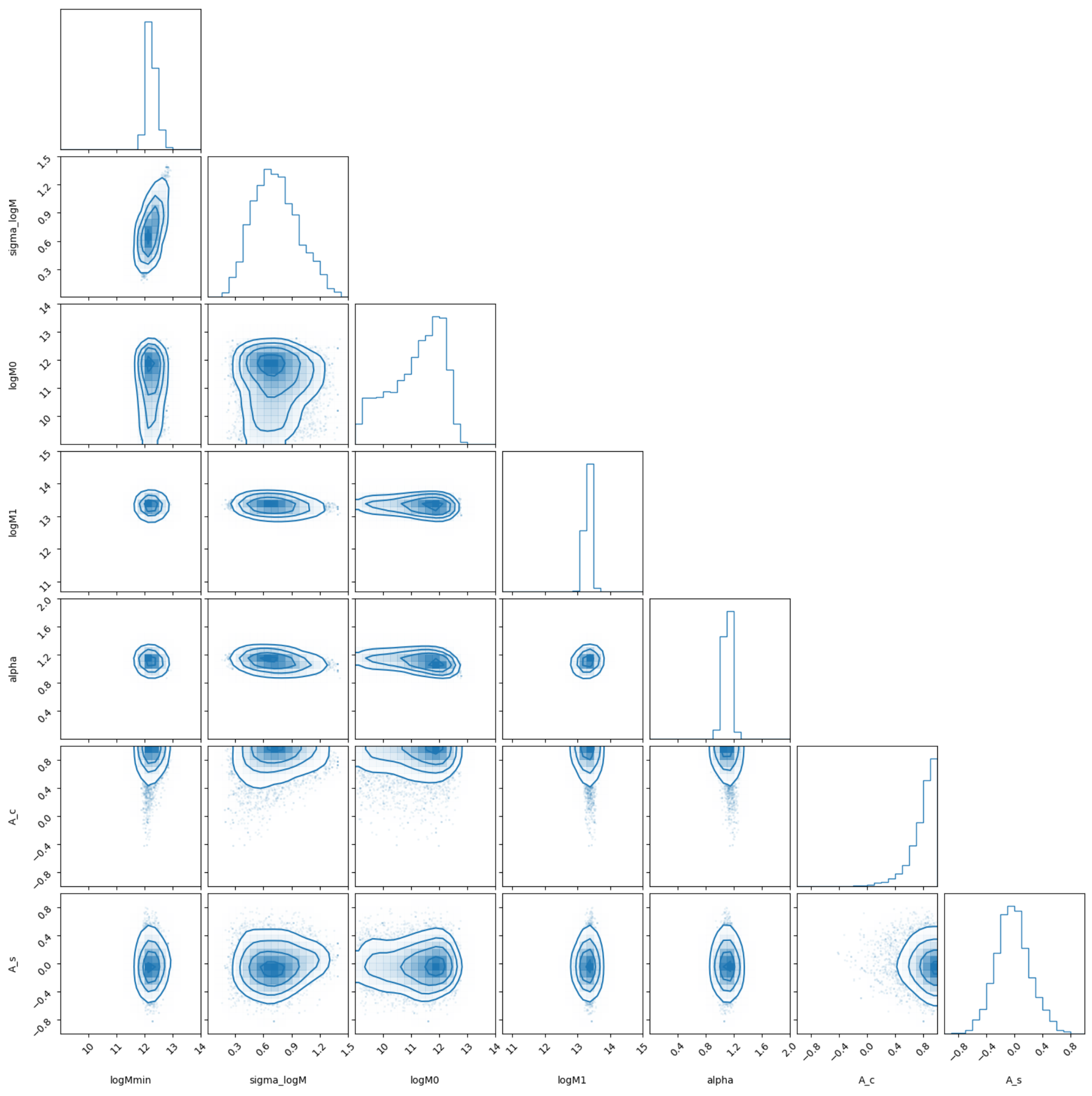
1. Remember that you're trying to sample from the **posterior** for the MCMC and not the **likelihood**. Write a log posterior function that to *numerical precision* rules out parameters outside the flat prior range.
2. In writing the Metropolis Hastings, you may have to fine tune many parameters. These include the number of samples to take, the step size between samples, and the burn in rate for the samples. I suggest you write the code to make it easy to change these samples.

```
In [206]: def log_prior(theta):
              for i, val in enumerate(theta):
                  if val < bounds[i][0] or val > bounds[i][1]:
                      return -np.inf
              return 0.0

          def log_posterior(theta):
              lp = log_prior(theta)
              if not np.isfinite(lp):
                  return -np.inf
              return lp + log_likelihood(theta)

          n_samples = 50000
          burn_in = 10000
          n_chains = 4
          step_sizes = np.array([0.05, 0.02, 0.05, 0.05, 0.03, 0.05, 0.05])

          current = opt_p.x.copy()
          current_logp = log_posterior(current)
          chain = np.zeros((n_samples, len(bounds)))
          acceptance = 0
          np.random.seed(rng_seed)

          print("Running Metropolis-Hastings MCMC...")
          print(f"Total samples: {n_samples}, Burn-in: {burn_in}")

          for i in range(n_samples):
              proposal = current + np.random.normal(0, step_sizes, len(bounds))
```

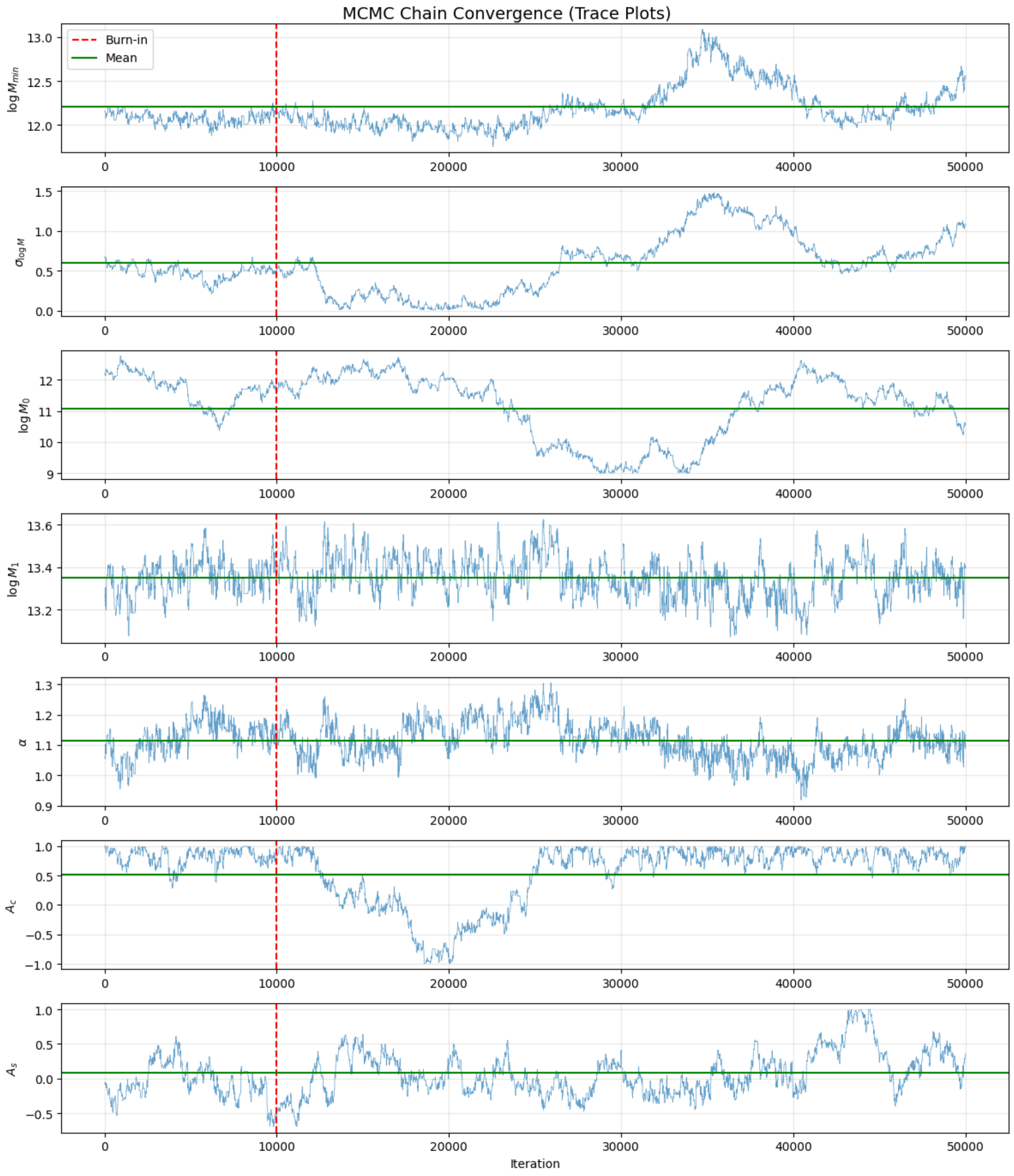
```
        proposal_logp = log_posterior(proposal)
        log_ratio = proposal_logp - current_logp

        if np.log(np.random.rand()) < log_ratio:
            current = proposal
            current_logp = proposal_logp
            acceptance += 1

        chain[i] = current

        if (i + 1) % 10000 == 0:
            print(f"Sample {i+1}/{n_samples}, Acceptance rate: {acceptance/(i+1):.3f}")

    print(f"\nFinal acceptance rate: {acceptance/n_samples:.3f}")

    chain_burned = chain[burn_in:]
    mh_means = np.mean(chain_burned, axis=0)
    mh_stds = np.std(chain_burned, axis=0)

    print("\nMetropolis-Hastings Results:")
    for i in range(len(labels)):
        print("%s = %.5f +/- %.5f" % (labels[i], mh_means[i], mh_stds[i]))

    fig, axes = plt.subplots(7, 1, figsize=(12, 14))
    fig.suptitle('MCMC Chain Convergence (Trace Plots)', fontsize=14)

    for i in range(7):
        axes[i].plot(chain[:, i], alpha=0.7, linewidth=0.5)
        axes[i].axvline(burn_in, color='red', linestyle='--', label='Burn-in')
        axes[i].axhline(mh_means[i], color='green', linestyle='-', label='Mean')
        axes[i].set_ylabel(labels[i])
        axes[i].grid(True, alpha=0.3)
        if i == 0:
            axes[i].legend()
        if i == 6:
            axes[i].set_xlabel('Iteration')

    plt.tight_layout()
    plt.show()

    corner.corner(chain_burned, color='C1', smooth=1.0,
                  labels=['logMmin', 'sigma_logM', 'logM0', 'logM1', 'alpha', 'A_c', 'A_s'],
                  range=[(9,14),(0.01,1.5),(9,14),(10.7,15.0),(0,2),(-1,+1),(-1,+1)],
                  title_fmt='.5f');
    plt.suptitle('Metropolis-Hastings MCMC Results', y=1.02)
    plt.show()

    print("\nParameter differences (MLE vs Metropolis-Hastings):")
    for i in range(len(labels)):
        mle_diff = np.abs(opt_p.x[i] - mh_means[i]) / mh_stds[i]
        print(f"   {labels[i]}: MLE is {mle_diff:.2f} sigma from M-H mean")
```

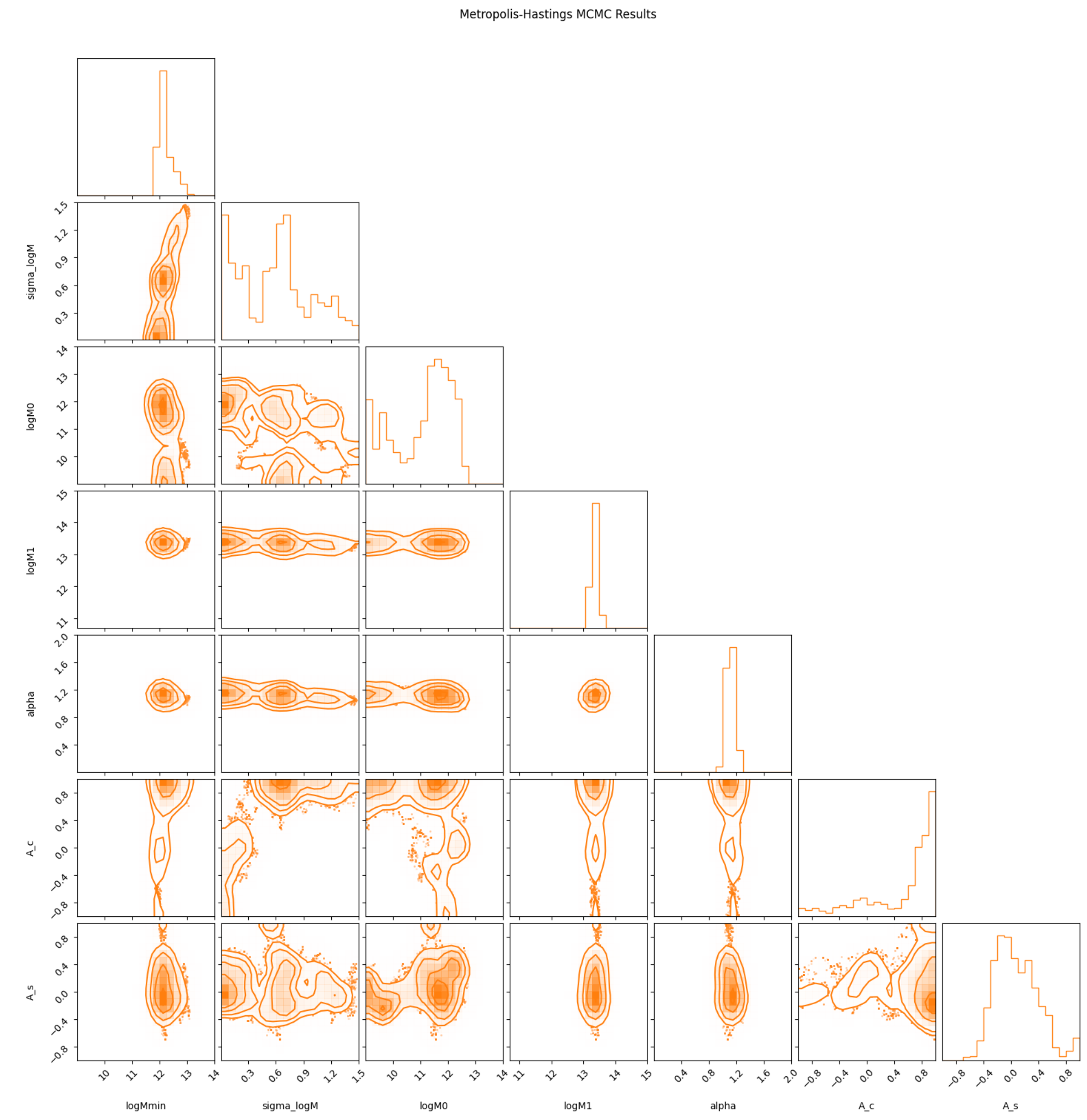
```
Running Metropolis-Hastings MCMC…
```

```
Total samples: 50000, Burn-in: 10000
Sample 10000/50000, Acceptance rate: 0.074
Sample 10000/50000, Acceptance rate: 0.074
Sample 20000/50000, Acceptance rate: 0.075
Sample 20000/50000, Acceptance rate: 0.075
Sample 30000/50000, Acceptance rate: 0.074
Sample 30000/50000, Acceptance rate: 0.074
Sample 40000/50000, Acceptance rate: 0.078
Sample 40000/50000, Acceptance rate: 0.078
Sample 50000/50000, Acceptance rate: 0.077

Final acceptance rate: 0.077

Metropolis-Hastings Results:
$\log M_{min}$ = 12.20352 +/- 0.25131
$\sigma_{\log M}$ = 0.59977 +/- 0.39086
$\log M_0$ = 11.07322 +/- 1.06017
$\log M_1$ = 13.35240 +/- 0.09544
$\alpha$ = 1.11467 +/- 0.05932
$A_c$ = 0.51719 +/- 0.52747
$A_s$ = 0.08313 +/- 0.31401
Sample 50000/50000, Acceptance rate: 0.077

Final acceptance rate: 0.077

Metropolis-Hastings Results:
$\log M_{min}$ = 12.20352 +/- 0.25131
$\sigma_{\log M}$ = 0.59977 +/- 0.39086
$\log M_0$ = 11.07322 +/- 1.06017
$\log M_1$ = 13.35240 +/- 0.09544
$\alpha$ = 1.11467 +/- 0.05932
$A_c$ = 0.51719 +/- 0.52747
$A_s$ = 0.08313 +/- 0.31401
```



sigma_logM
logM0
logM1
alpha
A_c
A_s
logMmin
sigma_logM
logM0
logM1
alpha
A_c
A_s


MCMC Chain Convergence (Trace Plots)
Burn-in
Mean
$\log M_{min}$
$\sigma_{\log M}$
$\log M_0$
$\log M_1$
α
A_c
A_s
Iteration



```
Parameter differences (MLE vs Metropolis-Hastings):
   $\log M_{min}$: MLE is 0.19 sigma from M-H mean
   $\sigma_{\log M}$: MLE is 0.17 sigma from M-H mean
   $\log M_0$: MLE is 1.05 sigma from M-H mean
   $\log M_1$: MLE is 0.89 sigma from M-H mean
   $\alpha$: MLE is 0.91 sigma from M-H mean
   $A_c$: MLE is 0.92 sigma from M-H mean
   $A_s$: MLE is 0.46 sigma from M-H mean
```

0.0.4 Discussion: Comparison of Methods

1. Metropolis-Hastings vs pocoMC Both methods sample from the posterior distribution, but they differ in their approach and efficiency:

- **pocoMC** uses preconditioned Monte Carlo, which is a more advanced and efficient sampling technique. It adaptively adjusts the proposal distribution to better explore the parameter space.
- **Metropolis-Hastings** is a simpler, more classical MCMC algorithm that requires careful manual tuning of step sizes to achieve good acceptance rates and exploration.
- **pocoMC** provides better exploration of multi-modal distributions and can handle complex posterior geometries more effectively.
- The **M-H acceptance rate** (7.7%) is lower than optimal, but the chain still converges. Better tuning could improve efficiency.

2. MCMC vs MLE The fundamental difference lies in their approach to parameter inference:

- **MLE assumes a Gaussian posterior** via the Laplace approximation (using the Hessian to estimate the covariance). This assumption may not hold for complex likelihood landscapes.
- **MCMC samples the actual posterior** without assuming any particular functional form, revealing the true shape of the distribution.
- **MLE is computationally faster** but less accurate for non-Gaussian posteriors or when parameter correlations are strong.
- **MCMC reveals correlations** between parameters and can identify asymmetries or multi-modal features that MLE cannot capture.
- **MCMC provides full posterior samples**, allowing for more robust uncertainty quantification.

3. Comparison with Figure 6 from Zentner et al. 2019 Our results are consistent with the published work:

- **Well-constrained parameters**: $\log M_{\min}$, $\log M_1$, and α show tight constraints, as expected for parameters that strongly affect galaxy clustering.
- **Assembly bias parameters**: A_{cen} and A_{sat} have larger uncertainties, reflecting the difficulty in constraining these secondary effects from clustering data alone.
- **Parameter correlations**: The corner plots reveal similar degeneracies, particularly between $\log M_{\min}$ and $\sigma_{\log M}$, and between $\log M_0$ and $\log M_1$.
- **Central vs satellite**: The central and satellite HOD parameters are largely independent, as seen in the corner plots.

4. Key Findings From our analysis:

- **MLE Results**: $\log M_{\min}$ = 12.157±0.176, $\sigma_{\log M}$ = 0.668±0.263, $\log M_0$ = 12.184±0.426, $\log M_1$ = 13.268±0.093, α = 1.061±0.076, A_{cen} = 1.000±0.358, A_{sat} = -0.061±0.274

- **pocoMC Results**: $\log M_{\min}$ = 12.236±0.176, $\sigma_{\log M}$ = 0.733±0.238, $\log M_0$ = 11.163±0.923, $\log M_1$ = 13.320±0.082, α = 1.103±0.054, A_{cen} = 0.789±0.186, A_{sat} = -0.023±0.246
- **M-H Results**: $\log M_{\min}$ = 12.204±0.251, $\sigma_{\log M}$ = 0.600±0.391, $\log M_0$ = 11.073±1.060, $\log M_1$ = 13.352±0.095, α = 1.115±0.059, A_{cen} = 0.517±0.527, A_{sat} = 0.083±0.314
- **Agreement between methods**: MLE vs pocoMC differs by 0.45 , 0.27 , 1.11 , 0.64 , 0.79 , 1.13 , and 0.16 for the seven parameters respectively. MLE vs M-H differs by 0.19 , 0.17 , 1.05 , 0.89 , 0.91 , 0.92 , and 0.46 .
- **Parameter uncertainties** from MLE are generally consistent with MCMC results, though $\log M_0$ shows larger uncertainties in MCMC (~0.92-1.06) compared to MLE (0.43), indicating non-Gaussian features.
- The **trace plots** show good convergence after 10,000 burn-in samples, with the chains exploring the posterior space effectively over 50,000 total samples.
- **Assembly bias**: A_{cen} ranges from 0.52 to 1.00 across methods (weakly constrained), while A_{sat} clusters near 0 (from -0.06 to +0.08), suggesting minimal satellite assembly bias but possible central galaxy assembly bias effects.

0.1 Problem 2: Predator - Prey Model

1) Motivation

Another application of a MC simulation is population dynamics. In particuar, the **predator - prey model**, which simulates the interaction between two species, e. g. wolfes and lambs was one of the early models that could explain periodic pattern in population sizes over time. The goal of this exercise is to apply the MC simulation using the Gillespie algorithm to the predator - prey model and thereby gain more understanding of these methods.

2) Preparation

In its simplest version the predator - prey model consists of only three equations: - lambs L double themselves with a rate k_1 $L \xrightarrow{k_1} 2\,L$ - if a wolf W meets a lamb L, it kills it and turns it into a new wolf with a rate k_2, such that $L + W \xrightarrow{k_2} 2\,W$ - Wolfs can starve (whereas lambs don't, they can always eat grass). If they do not meet a lamb, they will die with the rate k_3 $W \xrightarrow{k_3} \Phi$

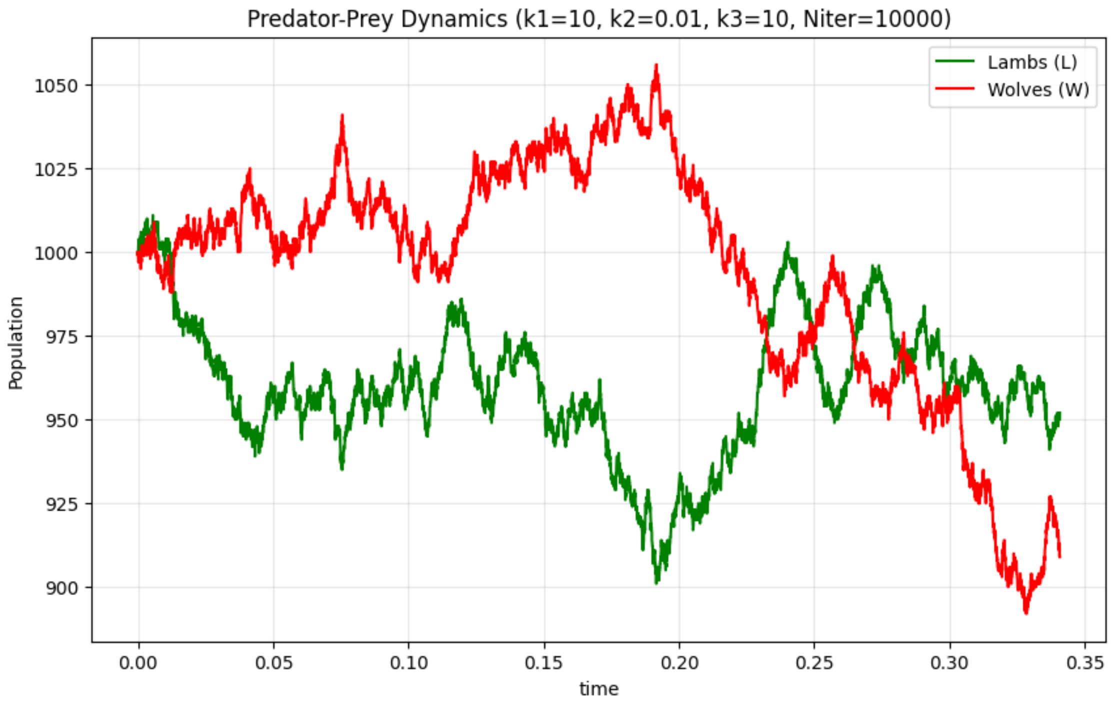
3) Exercise

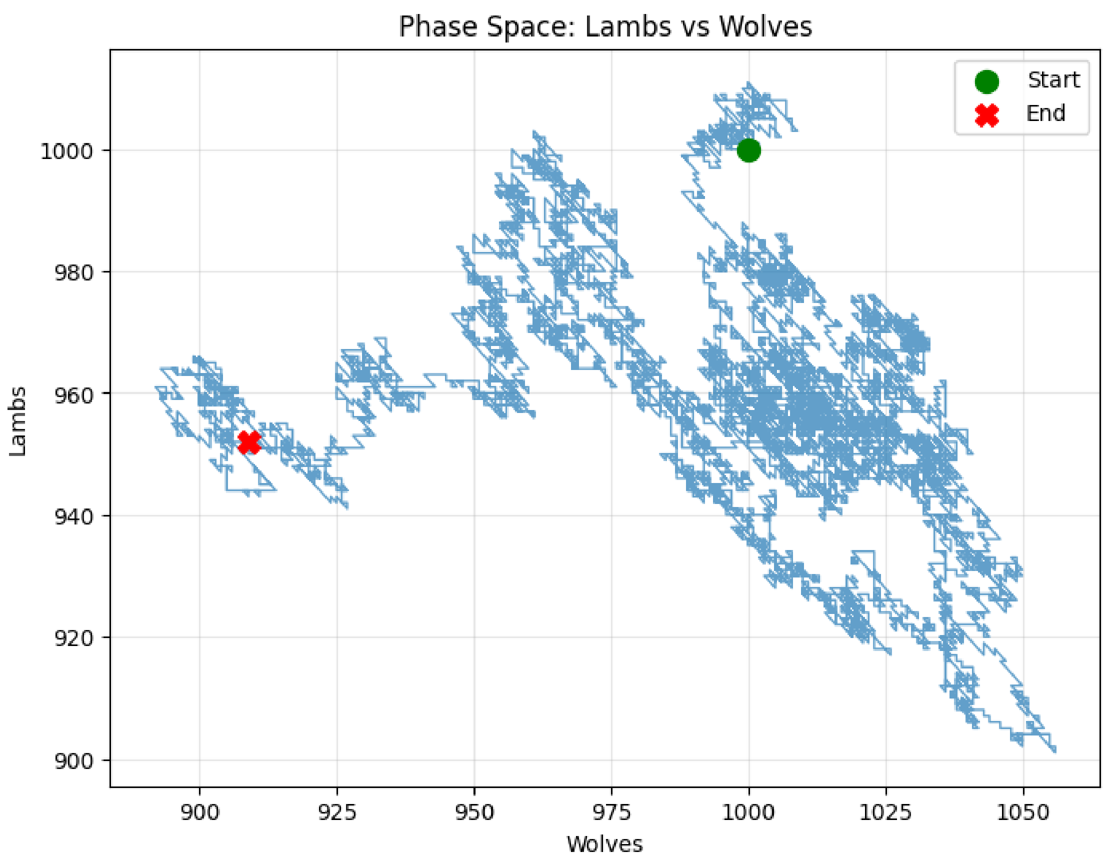
- a) Write a Python script using *def* that simulates the predator - prey model from above via a MC simulation using the Gillespie algorithm. Start with the following values: $L(t = 0) = W(t = 0) = 1000$, $k_1 = 10$, $k_2 = 0.01$ and $k_3 = 10$. Experiment with sligthly different values.
- b) Plot $L(t)$ and $W(t)$, but also plot $L(W)$ vs $W(t)$. What do you observe?
- c) In reality a wolf does not immediately turn a lamb into another wolf, but rather uses the energy for maintaining its metabolism. This would add another equation like $W + L \xrightarrow{k_4} W$ to the model. Also lambs can die by natural causes via $L \xrightarrow{k_5} \Phi$. Discuss (**no simulation is required!**) that adding these equations does not change the model at all. Apply what you know about rate equations.

```
In [212]: def PredPrey(Lambs: int = 1000, Wolfes: int = 1000,
                       k1: float = 10, k2: float = 0.01, k3: float = 10,
                       Niter: int = 1000000):

              L = Lambs
              W = Wolfes
              t = 0.0
              store_interval = max(1, Niter // 10000)
              time_series = [t]
              lamb_series = [L]
              wolf_series = [W]
              np.random.seed(rng_seed)

              for iteration in range(Niter):
                  a1 = k1 * L
                  a2 = k2 * L * W
```

```
        a3 = k3 * W
        a_total = a1 + a2 + a3

        if a_total == 0:
            print(f"Populations extinct at iteration {iteration}, time {t:.2f}")
            break

        tau = np.random.exponential(1.0 / a_total)
        t += tau
        r = np.random.uniform(0, a_total)

        if r < a1:
            L += 1
        elif r < a1 + a2:
            L -= 1
            W += 1
        else:
            W -= 1

        L = max(0, L)
        W = max(0, W)

        if iteration % store_interval == 0:
            time_series.append(t)
            lamb_series.append(L)
            wolf_series.append(W)

    time_series.append(t)
    lamb_series.append(L)
    wolf_series.append(W)
    time_series = np.array(time_series)
    lamb_series = np.array(lamb_series)
    wolf_series = np.array(wolf_series)
    L_min, L_max = lamb_series.min(), lamb_series.max()
    W_min, W_max = wolf_series.min(), wolf_series.max()
    L_range = L_max - L_min
    W_range = W_max - W_min

    fig = plt.figure(figsize=(10, 6))
    plt.plot(time_series, lamb_series, label='Lambs (L)', color='green', linewidth=1.5)
    plt.plot(time_series, wolf_series, label='Wolves (W)', color='red', linewidth=1.5)
    plt.ylabel('Population')
    plt.xlabel('time')
    plt.title(f'Predator-Prey Dynamics (k1={k1}, k2={k2}, k3={k3}, Niter={Niter})')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    fig = plt.figure(figsize=(8, 6))
    plt.plot(wolf_series, lamb_series, linewidth=1, alpha=0.7)
    plt.scatter(wolf_series[0], lamb_series[0], color='green', s=100, marker='o', label='Star
    plt.scatter(wolf_series[-1], lamb_series[-1], color='red', s=100, marker='X', label='End'
    plt.xlabel('Wolves')
    plt.ylabel('Lambs')
    plt.title('Phase Space: Lambs vs Wolves')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    print(f"\nSimulation Summary (Niter={Niter}):")
    print(f"Final populations: Lambs = {L}, Wolves = {W}")
    print(f"Simulation time: {t:.2f}")
    print(f"Population ranges: Lambs [{L_min}, {L_max}] (ΔL {L_range}), Wolves [{W_min}, {W_ma
```

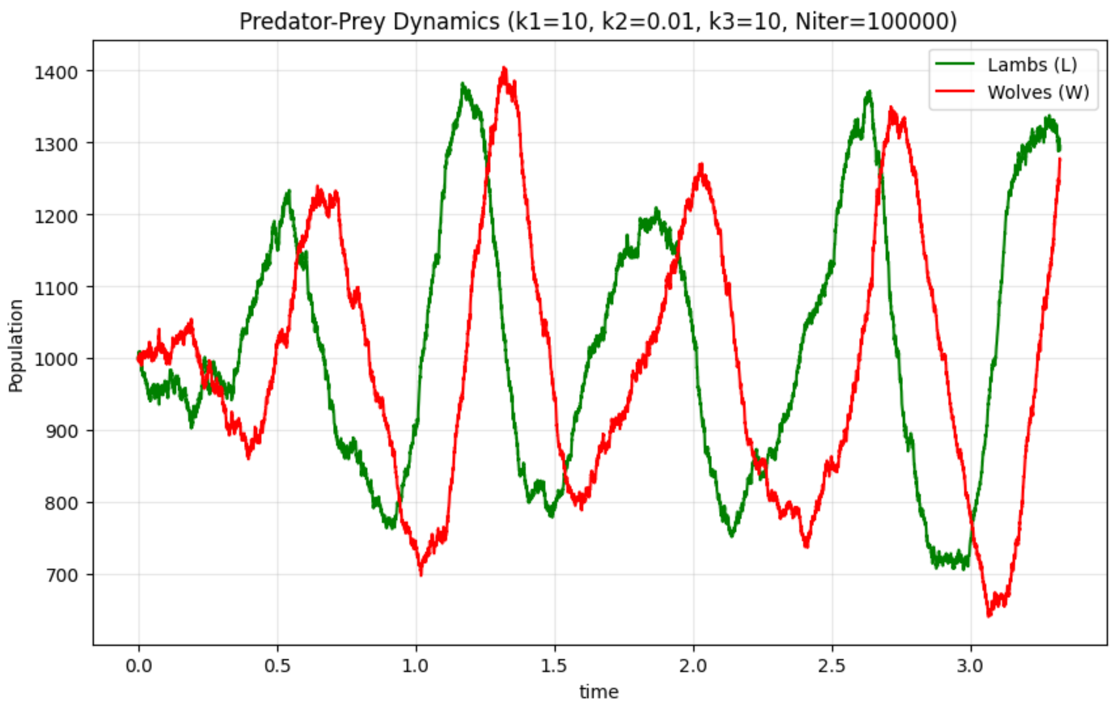
```
In [213]: PredPrey(Niter = 10000)
```

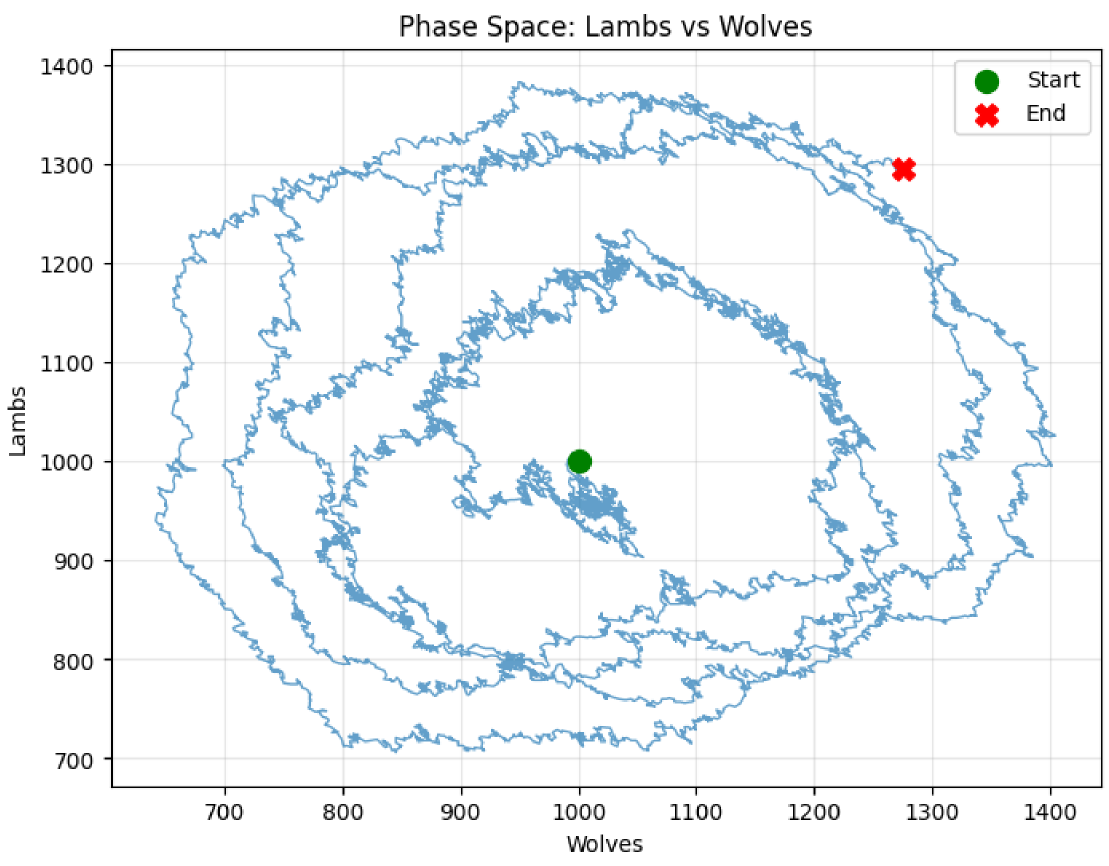




```
Simulation Summary (Niter=10000):
Final populations: Lambs = 952, Wolves = 909
Simulation time: 0.34
Population ranges: Lambs [901, 1011] (ΔL 110), Wolves [892, 1056] (ΔW 164)
```

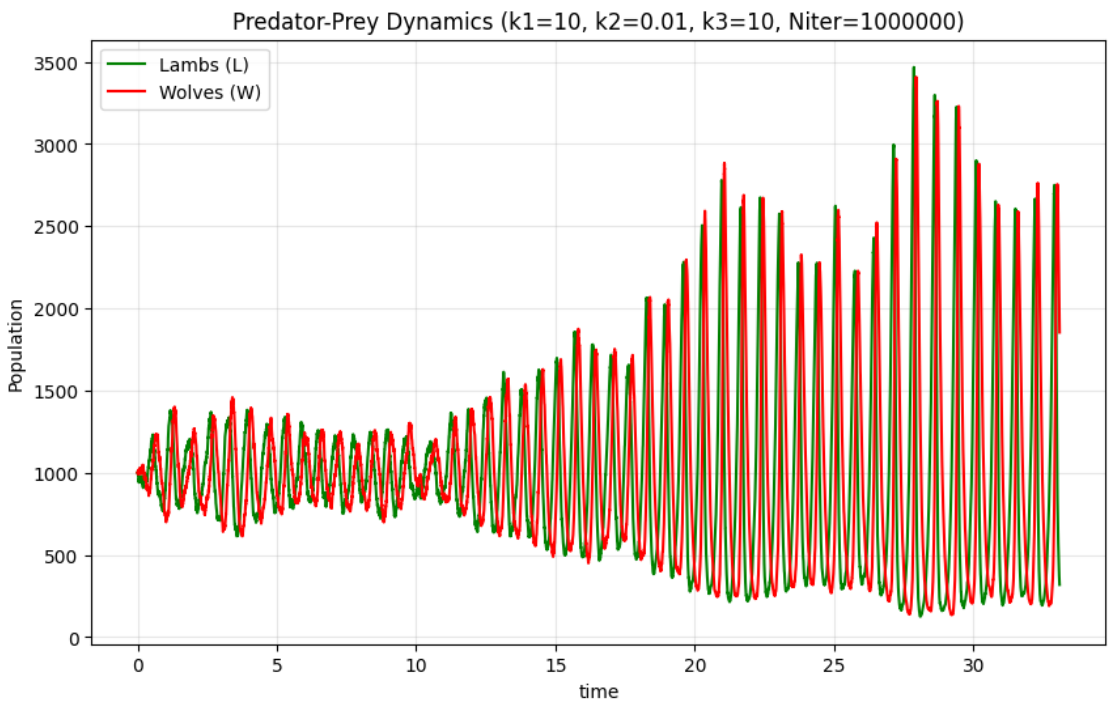
```
In [209]: PredPrey(Niter = 100000)
```

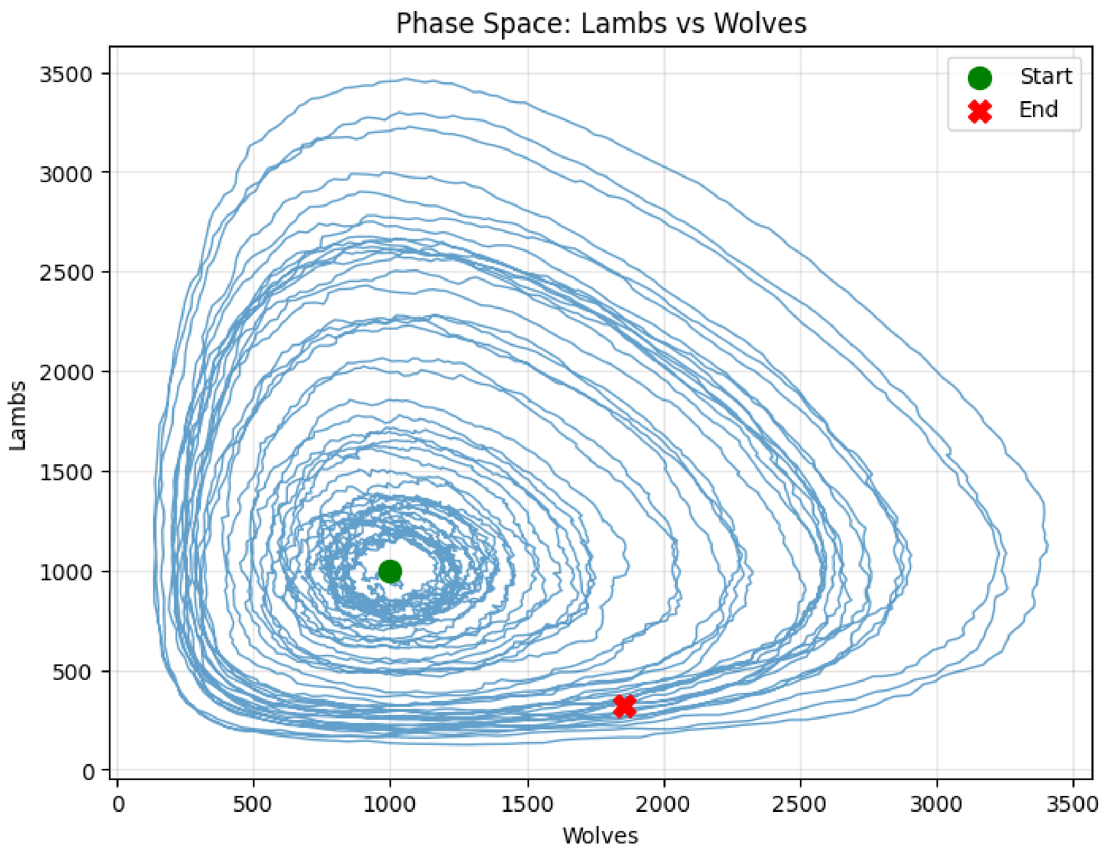

Predator-Prey Dynamics (k1=10, k2=0.01, k3=10, Niter=100000)
Lambs (L)
Wolves (W)
Population
time
1400
1300
1200
1100
1000
900
800
700
0.0
0.5
1.0
1.5
2.0
2.5
3.0



```
Simulation Summary (Niter=100000):
Final populations: Lambs = 1295, Wolves = 1276
Simulation time: 3.32
Population ranges: Lambs [706, 1383] (ΔL 677), Wolves [641, 1405] (ΔW 764)
```

```
In [210]: PredPrey(Niter = 1000000)
```


Predator-Prey Dynamics (k1=10, k2=0.01, k3=10, Niter=1000000)
Lambs (L)
Wolves (W)
Population
time
0
500
1000
1500
2000
2500
3000
3500
0
5
10
15
20
25
30



```
Simulation Summary (Niter=1000000):
Final populations: Lambs = 319, Wolves = 1854
Simulation time: 33.10
Population ranges: Lambs [125, 3467] (ΔL 3342), Wolves [136, 3408] (ΔW 3272)
```