Using ArviZ for MCMC Sampler Comparison

ArviZ provides a comprehensive framework for comparing MCMC samplers through standardized diagnostics, visualization, and data structures. (ArviZ +4) This guide demonstrates practical implementation for comparing four key MCMC algorithms: independent metropolis, metropolis-hastings random walk, gibbs sampler, and metropolis adjusted Langevin algorithm (MALA).

Structuring MCMC output for ArviZ analysis

ArviZ requires MCMC data in a standardized format using **InferenceData** objects built on xarray Datasets. (ArviZ +4) The core requirement is the **(chain, draw)** dimensional structure for all sampling-based variables. (ArviZ +5)

Essential data structure requirements

python

```python
import arviz as az
import numpy as np

# Standard MCMC output format for ArviZ
mcmc_results = {
    'posterior': {
        'param1': np.array(shape=(n_chains, n_draws)),
        'param2': np.array(shape=(n_chains, n_draws, param2_dims...)),
    },
    'sample_stats': {
        'lp': np.array(shape=(n_chains, n_draws)),      # log probability
        'accepted': np.array(shape=(n_chains, n_draws)),  # acceptance indicator
        'step_size': np.array(shape=(n_chains, n_draws)), # or scalar per chain
    },
    'observed_data': {
        'y': observed_data_array  # no chain/draw dimensions
    }
}
```

Universal conversion function

python

```python
def convert_mcmc_to_arviz(samples, log_prob, sample_stats=None,
                          coords=None, dims=None, var_names=None):
    """
    Convert custom MCMC output to ArviZ InferenceData

    Parameters:
    -----------
    samples : dict
        Parameter samples {param_name: array(n_chains, n_draws, ...)}
    log_prob : array
        Log probability values (n_chains, n_draws)
    sample_stats : dict, optional
        Additional sampler statistics
    coords : dict, optional
        Coordinate labels for dimensions
    dims : dict, optional
        Dimension mappings for variables
    """

    # Ensure proper dimensional structure
    posterior_data = {}
    for name, values in samples.items():
        values = np.array(values)
        if values.ndim == 1:  # Single chain: (draws,) → (1, draws)
            posterior_data[name] = values.reshape(1, -1)
        elif values.ndim == 2:
            if values.shape[0] > values.shape[1]:  # Likely (draws, params)
                posterior_data[name] = values.T.reshape(1, values.shape[0], -1)
            else:  # Already (chains, draws) or (chains, params)
                posterior_data[name] = values
        else:
            posterior_data[name] = values

    # Sample statistics with log probability
    stats_data = {'lp': log_prob}
    if sample_stats:
        stats_data.update(sample_stats)

    # Create InferenceData object
    idata = az.from_dict(
        posterior=posterior_data,
        sample_stats=stats_data,
        coords=coords,
        dims=dims
    )
```

```
    return idata
```

Four MCMC sampler implementations

Independent Metropolis sampler

```python
```

```python
class IndependentMetropolisSampler:
    def __init__(self, log_target, proposal_dist):
        self.log_target = log_target
        self.proposal_dist = proposal_dist

    def sample(self, n_samples, initial_value, n_chains=1):
        samples = []
        log_probs = []
        accept_rates = []

        for chain in range(n_chains):
            chain_samples = []
            chain_log_probs = []
            chain_accepts = 0

            current = initial_value.copy()
            current_log_prob = self.log_target(current)

            for i in range(n_samples):
                # Independent proposal
                proposal = self.proposal_dist.rvs()
                proposal_log_prob = self.log_target(proposal)

                # Acceptance ratio for independent proposals
                log_alpha = (proposal_log_prob + self.proposal_dist.logpdf(current) -
                             current_log_prob - self.proposal_dist.logpdf(proposal))

                if np.log(np.random.rand()) < log_alpha:
                    current = proposal
                    current_log_prob = proposal_log_prob
                    chain_accepts += 1

                chain_samples.append(current.copy())
                chain_log_probs.append(current_log_prob)

            samples.append(chain_samples)
            log_probs.append(chain_log_probs)
            accept_rates.append(chain_accepts / n_samples)

        return np.array(samples), np.array(log_probs), np.array(accept_rates)

# ArviZ integration
def independent_metropolis_to_arviz(samples, log_probs, accept_rates, var_names):
    """Convert Independent Metropolis output to ArviZ format"""

    # Structure posterior samples
```

```python
    posterior_dict = {}
    if len(var_names) == 1:
        posterior_dict[var_names[0]] = samples
    else:
        for i, name in enumerate(var_names):
            posterior_dict[name] = samples[:, :, i]

    # Sample statistics
    n_chains, n_draws = samples.shape[:2]
    sample_stats_dict = {
        'lp': log_probs,
        'accepted': np.ones((n_chains, n_draws)),  # All attempts tracked
        'accept_rate': np.broadcast_to(accept_rates.reshape(-1, 1), (n_chains, n_draws))
    }

    return az.from_dict(posterior=posterior_dict, sample_stats=sample_stats_dict)
```

Metropolis-Hastings Random Walk sampler

python

```python
class MetropolisHastingsSampler:
    def __init__(self, log_target, proposal_cov):
        self.log_target = log_target
        self.proposal_cov = proposal_cov

    def sample(self, n_samples, initial_value, n_chains=1):
        samples = []
        log_probs = []
        accept_rates = []

        for chain in range(n_chains):
            chain_samples = []
            chain_log_probs = []
            chain_accepts = 0

            current = initial_value.copy()
            current_log_prob = self.log_target(current)

            for i in range(n_samples):
                # Random walk proposal
                proposal = current + np.random.multivariate_normal(
                    np.zeros(len(current)), self.proposal_cov
                )
                proposal_log_prob = self.log_target(proposal)

                # Symmetric proposal → simplified acceptance ratio
                log_alpha = proposal_log_prob - current_log_prob

                if np.log(np.random.rand()) < log_alpha:
                    current = proposal
                    current_log_prob = proposal_log_prob
                    chain_accepts += 1

                chain_samples.append(current.copy())
                chain_log_probs.append(current_log_prob)

            samples.append(chain_samples)
            log_probs.append(chain_log_probs)
            accept_rates.append(chain_accepts / n_samples)

        return np.array(samples), np.array(log_probs), np.array(accept_rates)
```

Gibbs sampler

```python
```

```
class GibbsSampler:
    def __init__(self, conditional_samplers, log_target):
        self.conditional_samplers = conditional_samplers
        self.log_target = log_target

    def sample(self, n_samples, initial_value, n_chains=1):
        samples = []
        log_probs = []

        for chain in range(n_chains):
            chain_samples = []
            chain_log_probs = []

            current = initial_value.copy()

            for i in range(n_samples):
                # Update each parameter conditionally
                for j, sampler_func in enumerate(self.conditional_samplers):
                    current[j] = sampler_func(current, j)

                chain_samples.append(current.copy())
                chain_log_probs.append(self.log_target(current))

            samples.append(chain_samples)
            log_probs.append(chain_log_probs)

        return np.array(samples), np.array(log_probs)

# ArviZ integration - Gibbs has perfect acceptance
def gibbs_to_arviz(samples, log_probs, var_names):
    """Convert Gibbs sampler output to ArviZ format"""

    posterior_dict = {}
    for i, name in enumerate(var_names):
        posterior_dict[name] = samples[:, :, i]

    # Gibbs always accepts, no step size parameters
    sample_stats_dict = {
        'lp': log_probs,
        'accept_rate': np.ones_like(log_probs)  # Always 1.0 for Gibbs
    }

    return az.from_dict(posterior=posterior_dict, sample_stats=sample_stats_dict)
```

MALA (Metropolis Adjusted Langevin Algorithm) sampler

python

```python
class MALASampler:
    def __init__(self, log_target, grad_log_target, step_size=0.1):
        self.log_target = log_target
        self.grad_log_target = grad_log_target
        self.step_size = step_size

    def sample(self, n_samples, initial_value, n_chains=1):
        samples = []
        log_probs = []
        accept_rates = []
        grad_norms = []

        for chain in range(n_chains):
            chain_samples = []
            chain_log_probs = []
            chain_grad_norms = []
            chain_accepts = 0

            current = initial_value.copy()
            current_log_prob = self.log_target(current)
            current_grad = self.grad_log_target(current)

            for i in range(n_samples):
                # MALA proposal using gradient information
                drift = current + 0.5 * self.step_size**2 * current_grad
                proposal = drift + self.step_size * np.random.standard_normal(len(current))

                proposal_log_prob = self.log_target(proposal)
                proposal_grad = self.grad_log_target(proposal)

                # Calculate acceptance probability for MALA
                proposal_to_current = current - proposal - 0.5 * self.step_size**2 * proposal_grad
                current_to_proposal = proposal - current - 0.5 * self.step_size**2 * current_grad

                log_alpha = (proposal_log_prob - current_log_prob +
                        -0.25 * np.sum(proposal_to_current**2) / self.step_size**2 +
                        0.25 * np.sum(current_to_proposal**2) / self.step_size**2)

                if np.log(np.random.rand()) < log_alpha:
                    current = proposal
                    current_log_prob = proposal_log_prob
                    current_grad = proposal_grad
                    chain_accepts += 1

                chain_samples.append(current.copy())
                chain_log_probs.append(current_log_prob)
```

```
            chain_grad_norms.append(np.linalg.norm(current_grad))

        samples.append(chain_samples)
        log_probs.append(chain_log_probs)
        grad_norms.append(chain_grad_norms)
        accept_rates.append(chain_accepts / n_samples)

    return np.array(samples), np.array(log_probs), np.array(accept_rates), np.array(grad_norms)

# ArviZ integration for MALA
def mala_to_arviz(samples, log_probs, accept_rates, grad_norms, step_size, var_names):
    """Convert MALA sampler output to ArviZ format"""

    posterior_dict = {}
    for i, name in enumerate(var_names):
        posterior_dict[name] = samples[:, :, i]

    n_chains, n_draws = samples.shape[:2]
    sample_stats_dict = {
        'lp': log_probs,
        'accept_rate': np.broadcast_to(accept_rates.reshape(-1, 1), (n_chains, n_draws)),
        'step_size': np.full((n_chains, n_draws), step_size),
        'grad_norm': grad_norms
    }

    return az.from_dict(posterior=posterior_dict, sample_stats=sample_stats_dict)
```

Key ArviZ diagnostic functions

Essential MCMC diagnostics

python

```python
def comprehensive_diagnostics(idata, var_names=None):
    """Calculate all major MCMC diagnostics using ArviZ"""
    diagnostics = {}

    # Effective Sample Size (bulk and tail)
    diagnostics['ess_bulk'] = az.ess(idata, var_names=var_names, method='bulk')
    diagnostics['ess_tail'] = az.ess(idata, var_names=var_names, method='tail')

    # R-hat convergence diagnostic (rank-normalized)
    diagnostics['rhat'] = az.rhat(idata, var_names=var_names, method='rank')

    # Monte Carlo Standard Error
    diagnostics['mcse_mean'] = az.mcse(idata, var_names=var_names, method='mean')
    diagnostics['mcse_sd'] = az.mcse(idata, var_names=var_names, method='sd')

    # Autocorrelation function
    posterior_samples = az.extract(idata, var_names=var_names)
    diagnostics['autocorr'] = {
        var: az.autocorr(posterior_samples[var].values) for var in posterior_samples.data_vars
    }

    # Comprehensive summary
    diagnostics['summary'] = az.summary(idata, var_names=var_names, stat_focus='convergence')

    return diagnostics

# Interpretation guidelines for diagnostics
def interpret_diagnostics(diagnostics):
    """Provide interpretation of diagnostic results"""
    interpretation = {}

    # R-hat interpretation
    rhat_vals = diagnostics['rhat'].to_array().values.flatten()
    max_rhat = np.max(rhat_vals)
    interpretation['convergence'] = {
        'max_rhat': max_rhat,
        'status': 'excellent' if max_rhat < 1.01 else
                  'good' if max_rhat < 1.05 else
                  'poor' if max_rhat < 1.1 else 'very_poor'
    }

    # ESS interpretation
    ess_bulk_vals = diagnostics['ess_bulk'].to_array().values.flatten()
    min_ess_bulk = np.min(ess_bulk_vals)
    interpretation['efficiency'] = {
        'min_ess_bulk': min_ess_bulk,
```

```
        'status': 'excellent' if min_ess_bulk > 1000 else
                  'good' if min_ess_bulk > 400 else
                  'adequate' if min_ess_bulk > 100 else 'poor'
    }

    return interpretation
```

Specialized diagnostic metrics

python

```python
# Calculate efficiency ratios between samplers
def sampler_efficiency_comparison(idata_dict):
    """Compare sampling efficiency across multiple samplers"""
    efficiency_metrics = {}

    for sampler_name, idata in idata_dict.items():
        # Basic efficiency metrics
        ess_bulk = az.ess(idata, method='bulk')
        ess_tail = az.ess(idata, method='tail')
        rhat = az.rhat(idata)

        n_chains = idata.posterior.dims['chain']
        n_draws = idata.posterior.dims['draw']
        total_samples = n_chains * n_draws

        # Efficiency ratios
        bulk_efficiency = ess_bulk.to_array().min() / total_samples
        tail_efficiency = ess_tail.to_array().min() / total_samples

        # Convergence assessment
        max_rhat = float(rhat.to_array().max())
        converged = max_rhat < 1.05

        efficiency_metrics[sampler_name] = {
            'total_samples': total_samples,
            'min_ess_bulk': float(ess_bulk.to_array().min()),
            'min_ess_tail': float(ess_tail.to_array().min()),
            'bulk_efficiency': float(bulk_efficiency),
            'tail_efficiency': float(tail_efficiency),
            'max_rhat': max_rhat,
            'converged': converged
        }

        # Add acceptance rate if available
        if hasattr(idata, 'sample_stats') and 'accept_rate' in idata.sample_stats:
            efficiency_metrics[sampler_name]['mean_accept_rate'] = float(
                idata.sample_stats['accept_rate'].mean()
            )

    return efficiency_metrics
```

Visualization techniques for sampler comparison

Core diagnostic plots

```python
```

```
import matplotlib.pyplot as plt

def create_comparison_plots(idata_dict, var_names=None):
    """Create comprehensive comparison plots for multiple samplers"""

    n_samplers = len(idata_dict)
    sampler_names = list(idata_dict.keys())

    # 1. Trace plot comparison
    fig, axes = plt.subplots(n_samplers, 2, figsize=(12, 4*n_samplers))
    if n_samplers == 1:
        axes = axes.reshape(1, -1)

    for i, (name, idata) in enumerate(idata_dict.items()):
        az.plot_trace(idata, var_names=var_names, ax=axes[i])
        axes[i, 0].set_title(f'{name} - Marginal Distributions')
        axes[i, 1].set_title(f'{name} - Trace Evolution')

    plt.tight_layout()
    plt.suptitle('Trace Plot Comparison', y=1.02, size=16)
    plt.show()

    # 2. Rank plot comparison (more sensitive than trace plots)
    fig, axes = plt.subplots(1, n_samplers, figsize=(4*n_samplers, 6))
    if n_samplers == 1:
        axes = [axes]

    for i, (name, idata) in enumerate(idata_dict.items()):
        az.plot_rank(idata, var_names=var_names, ax=axes[i])
        axes[i].set_title(f'{name}\nRank Plot')

    plt.tight_layout()
    plt.show()

    # 3. ESS comparison plot
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Bulk ESS
    ess_data = {}
    for name, idata in idata_dict.items():
        ess_bulk = az.ess(idata, var_names=var_names, method='bulk')
        ess_data[name] = ess_bulk.to_array().values.flatten()

    # Box plot of ESS values
    axes[0].boxplot(list(ess_data.values()), labels=list(ess_data.keys()))
    axes[0].set_ylabel('Effective Sample Size (Bulk)')
```

```
    axes[0].set_title('Bulk ESS Comparison')
    axes[0].axhline(y=400, color='red', linestyle='--', alpha=0.7, label='Recommended minimum')
    axes[0].legend()

    # Tail ESS
    ess_tail_data = {}
    for name, idata in idata_dict.items():
        ess_tail = az.ess(idata, var_names=var_names, method='tail')
        ess_tail_data[name] = ess_tail.to_array().values.flatten()

    axes[1].boxplot(list(ess_tail_data.values()), labels=list(ess_tail_data.keys()))
    axes[1].set_ylabel('Effective Sample Size (Tail)')
    axes[1].set_title('Tail ESS Comparison')
    axes[1].axhline(y=100, color='red', linestyle='--', alpha=0.7, label='Recommended minimum')
    axes[1].legend()

    plt.tight_layout()
    plt.show()

    # 4. Autocorrelation comparison
    fig, axes = plt.subplots(1, n_samplers, figsize=(4*n_samplers, 5))
    if n_samplers == 1:
        axes = [axes]

    for i, (name, idata) in enumerate(idata_dict.items()):
        az.plot_autocorr(idata, var_names=var_names, ax=axes[i], max_lag=50)
        axes[i].set_title(f'{name}\nAutocorrelation')

    plt.tight_layout()
    plt.show()

# Advanced diagnostic visualization
def plot_diagnostic_summary(efficiency_metrics):
    """Create summary visualization of efficiency metrics"""

    samplers = list(efficiency_metrics.keys())
    metrics = ['bulk_efficiency', 'tail_efficiency', 'max_rhat', 'mean_accept_rate']

    fig, axes = plt.subplots(2, 2, figsize=(12, 8))
    axes = axes.flatten()

    for i, metric in enumerate(metrics):
        if metric in efficiency_metrics[samplers[0]]:  # Check if metric exists
            values = [efficiency_metrics[sampler].get(metric, 0) for sampler in samplers]

            bars = axes[i].bar(samplers, values)
            axes[i].set_title(metric.replace('_', ' ').title())
```

```
        axes[i].tick_params(axis='x', rotation=45)

        # Add reference lines for some metrics
        if metric == 'max_rhat':
            axes[i].axhline(y=1.01, color='green', linestyle='--', alpha=0.7, label='Excellent')
            axes[i].axhline(y=1.05, color='orange', linestyle='--', alpha=0.7, label='Good')
            axes[i].axhline(y=1.1, color='red', linestyle='--', alpha=0.7, label='Poor')
            axes[i].legend()
        elif 'efficiency' in metric:
            axes[i].set_ylabel('ESS / Total Samples')

        # Color bars based on performance
        if metric == 'max_rhat':
            for j, (bar, val) in enumerate(zip(bars, values)):
                if val < 1.01:
                    bar.set_color('green')
                elif val < 1.05:
                    bar.set_color('orange')
                else:
                    bar.set_color('red')

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

Specialized comparison plots

python

```python
def plot_mixing_comparison(idata_dict, param_name='theta'):
    """Compare chain mixing across samplers for a specific parameter"""

    fig, axes = plt.subplots(len(idata_dict), 1, figsize=(12, 3*len(idata_dict)))
    if len(idata_dict) == 1:
        axes = [axes]

    for i, (sampler_name, idata) in enumerate(idata_dict.items()):
        if param_name in idata.posterior:
            # Plot each chain separately to visualize mixing
            chains = idata.posterior[param_name]
            for chain_idx in range(chains.dims['chain']):
                axes[i].plot(chains.isel(chain=chain_idx), alpha=0.7,
                             label=f'Chain {chain_idx+1}')

            axes[i].set_title(f'{sampler_name} - {param_name} Chain Mixing')
            axes[i].set_xlabel('Iteration')
            axes[i].set_ylabel(param_name)
            axes[i].legend()

    plt.tight_layout()
    plt.show()

def plot_posterior_comparison(idata_dict, var_names=None):
    """Compare posterior distributions across samplers"""

    n_vars = len(var_names) if var_names else len(list(idata_dict.values())[0].posterior.data_vars)
    n_samplers = len(idata_dict)

    fig, axes = plt.subplots(n_vars, 1, figsize=(10, 3*n_vars))
    if n_vars == 1:
        axes = [axes]

    # Get variable names
    if not var_names:
        var_names = list(list(idata_dict.values())[0].posterior.data_vars)

    colors = plt.cm.tab10(np.linspace(0, 1, n_samplers))

    for i, var_name in enumerate(var_names):
        for j, (sampler_name, idata) in enumerate(idata_dict.items()):
            if var_name in idata.posterior:
                samples = az.extract(idata.posterior[var_name])
                axes[i].hist(samples.values.flatten(), bins=50, alpha=0.6,
                             color=colors[j], label=sampler_name, density=True)
```

```
    axes[i].set_title(f'Posterior Distribution: {var_name}')
    axes[i].set_xlabel(var_name)
    axes[i].set_ylabel('Density')
    axes[i].legend()

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

Best practices for fair MCMC comparison

Standardized comparison framework

python

```
def fair_sampler_comparison(samplers_dict, target_func, initial_value,
                            n_samples=2000, n_chains=4, var_names=None):
    """
    Conduct fair comparison between MCMC samplers

    Parameters:
    -----------
    samplers_dict : dict
        Dictionary of {name: sampler_instance}
    target_func : callable
        Log target distribution function
    initial_value : array
        Starting values for all samplers
    n_samples : int
        Number of samples per chain
    n_chains : int
        Number of chains to run
    var_names : list, optional
        Parameter names for ArviZ
    """

    results = {}
    runtimes = {}

    # Set same random seed for all samplers for reproducibility
    base_seed = 42

    for sampler_name, sampler in samplers_dict.items():
        print(f"Running {sampler_name}...")
        np.random.seed(base_seed)

        start_time = time.time()

        # Run sampler (specific to sampler type)
        if hasattr(sampler, 'sample'):
            if sampler_name.lower() == 'gibbs':
                samples, log_probs = sampler.sample(n_samples, initial_value, n_chains)
                idata = gibbs_to_arviz(samples, log_probs, var_names)
            elif sampler_name.lower() == 'mala':
                samples, log_probs, accept_rates, grad_norms = sampler.sample(
                    n_samples, initial_value, n_chains)
                idata = mala_to_arviz(samples, log_probs, accept_rates,
                                      grad_norms, sampler.step_size, var_names)
            else:  # Metropolis variants
                samples, log_probs, accept_rates = sampler.sample(
                    n_samples, initial_value, n_chains)
```

```
            idata = convert_mcmc_to_arviz(
                {name: samples[:, :, i] for i, name in enumerate(var_names)},
                log_probs,
                {'accept_rate': np.broadcast_to(accept_rates.reshape(-1, 1),
                                (n_chains, n_samples))}
            )

        end_time = time.time()
        runtimes[sampler_name] = end_time - start_time
        results[sampler_name] = idata

        print(f"  Completed in {runtimes[sampler_name]:.2f} seconds")

    # Calculate comprehensive diagnostics
    efficiency_metrics = sampler_efficiency_comparison(results)

    # Add runtime information
    for sampler_name in efficiency_metrics:
        efficiency_metrics[sampler_name]['runtime'] = runtimes[sampler_name]
        efficiency_metrics[sampler_name]['samples_per_second'] = (
            n_chains * n_samples / runtimes[sampler_name]
        )

    return results, efficiency_metrics

# Comparison criteria
def rank_samplers(efficiency_metrics):
    """Rank samplers based on multiple criteria"""

    ranking_criteria = {
        'convergence': lambda x: -x['max_rhat'],  # Lower R-hat is better
        'bulk_efficiency': lambda x: x['bulk_efficiency'],  # Higher is better
        'tail_efficiency': lambda x: x['tail_efficiency'],  # Higher is better
        'speed': lambda x: x['samples_per_second'],  # Higher is better
    }

    rankings = {}
    for criterion, score_func in ranking_criteria.items():
        sorted_samplers = sorted(efficiency_metrics.items(),
                         key=lambda x: score_func(x[1]), reverse=True)
        rankings[criterion] = [name for name, _ in sorted_samplers]

    # Calculate overall ranking (simple average of ranks)
    overall_scores = {}
    for sampler_name in efficiency_metrics:
        rank_sum = 0
        for criterion_rankings in rankings.values():
```

```
            rank_sum += criterion_rankings.index(sampler_name)
        overall_scores[sampler_name] = rank_sum / len(rankings)

    overall_ranking = sorted(overall_scores.items(), key=lambda x: x[1])
    rankings['overall'] = [name for name, _ in overall_ranking]

    return rankings
```

Complete working example

python

```python
import scipy.stats as stats
import time

# Define test problem: Bayesian linear regression
def setup_test_problem():
    """Set up a standard test problem for sampler comparison"""
    np.random.seed(123)

    # Generate synthetic data
    n_obs = 50
    true_intercept, true_slope, true_sigma = 1.5, -0.8, 0.3

    x = np.random.uniform(-2, 2, n_obs)
    y = true_intercept + true_slope * x + np.random.normal(0, true_sigma, n_obs)

    # Define log posterior
    def log_posterior(params):
        intercept, slope, log_sigma = params
        sigma = np.exp(log_sigma)

        # Weakly informative priors
        prior = (stats.norm.logpdf(intercept, 0, 10) +
                 stats.norm.logpdf(slope, 0, 10) +
                 stats.norm.logpdf(log_sigma, 0, 1))

        # Likelihood
        mu = intercept + slope * x
        likelihood = np.sum(stats.norm.logpdf(y, mu, sigma))

        return prior + likelihood

    # Gradient for MALA
    def grad_log_posterior(params):
        intercept, slope, log_sigma = params
        sigma = np.exp(log_sigma)

        mu = intercept + slope * x
        residuals = y - mu

        # Gradient components
        grad_intercept = -intercept/100 + np.sum(residuals) / (sigma**2)
        grad_slope = -slope/100 + np.sum(residuals * x) / (sigma**2)
        grad_log_sigma = -log_sigma + len(y) - np.sum(residuals**2) / (sigma**2)

        return np.array([grad_intercept, grad_slope, grad_log_sigma])
```

```
    return log_posterior, grad_log_posterior, x, y, (true_intercept, true_slope, true_sigma)

# Run complete comparison
def run_complete_comparison():
    """Execute complete sampler comparison workflow"""

    # Setup problem
    log_post, grad_log_post, x_data, y_data, true_params = setup_test_problem()
    initial_value = np.array([0.0, 0.0, 0.0])  # intercept, slope, log_sigma
    var_names = ['intercept', 'slope', 'log_sigma']

    # Initialize samplers
    samplers = {
        'Metropolis-Hastings': MetropolisHastingsSampler(log_post, np.eye(3) * 0.05),
        'Independent Metropolis': IndependentMetropolisSampler(
            log_post,
            stats.multivariate_normal(mean=[1, -1, -1], cov=np.eye(3) * 0.5)
        ),
        'MALA': MALASampler(log_post, grad_log_post, step_size=0.1),
        'Gibbs': None  # Would need conditional samplers - simplified here
    }

    # Remove Gibbs for this example (requires conditional distributions)
    del samplers['Gibbs']

    # Run comparison
    print("=== Running MCMC Sampler Comparison ===")
    results, efficiency = fair_sampler_comparison(
        samplers, log_post, initial_value,
        n_samples=2000, n_chains=4, var_names=var_names
    )

    # Analyze results
    rankings = rank_samplers(efficiency)

    # Print summary
    print("\n=== Comparison Results ===")
    for i, sampler_name in enumerate(rankings['overall']):
        metrics = efficiency[sampler_name]
        print(f"\n{i+1}. {sampler_name}:")
        print(f"   Convergence (max R-hat): {metrics['max_rhat']:.4f}")
        print(f"   Bulk efficiency: {metrics['bulk_efficiency']:.4f}")
        print(f"   Runtime: {metrics['runtime']:.2f}s")
        print(f"   Samples/second: {metrics['samples_per_second']:.0f}")
        if 'mean_accept_rate' in metrics:
            print(f"   Acceptance rate: {metrics['mean_accept_rate']:.3f}")
```

```
    # Generate plots
    create_comparison_plots(results, var_names)
    plot_diagnostic_summary(efficiency)

    return results, efficiency, rankings

if __name__ == "__main__":
    results, efficiency, rankings = run_complete_comparison()
```

This comprehensive guide provides practical tools for comparing MCMC samplers using ArviZ. The framework emphasizes standardized data structures, robust diagnostics, and fair comparison methodology while maintaining flexibility for different sampler implementations. **Key success factors** include proper dimensional structuring, comprehensive diagnostic evaluation, and systematic comparison protocols that account for both statistical performance and computational efficiency.