# Lesson 7: Exercises - Forecasting Ebola

Aaron A. King Edward L. Ionides Translated in pypomp by Kunyang He

2025-12-24

# Table of contents I

# Table of contents II

This document contains worked solutions to the exercises from Lesson 7 on forecasting Ebola, implemented using **pypomp**.

# Import Packages

```
import jax
import jax.numpy as jnp
import jax.random as jr
import jax.scipy as jsp
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pypomp import Pomp
from pypomp.util import logmeanexp, logmeanexp_se
from scipy.stats import chi2

np.random.seed(594709947)
```

# Load Data and Build Model I

```
# Load the data
dat = pd.read_csv("https://kingaa.github.io/sbied/ebola/ebola_data.csv")

# Population sizes
populations = {
    "Guinea": 10628972.0,
    "Liberia": 4092310.0,
    "SierraLeone": 6190280.0
}

print(f"Data shape: {dat.shape}")
print(f"Countries: {dat['country'].unique()}")
```

```
Data shape: (74, 4)
Countries: ['Guinea' 'Liberia' 'SierraLeone']
```

# Load Data and Build Model II

```
# Model configuration
nstageE = 3  # Number of exposed stages (linear chain trick)
timestep = 0.1  # Euler integration timestep

# State names (global for reference)
statenames = (
    ["S"] +
    [f"E{i+1}" for i in range(nstageE)] +
    ["I", "R", "N_EI", "N_IR"]
)
print(f"State names: {statenames}")
print(f"N_EI is at index: {statenames.index('N_EI')}")
```

```
State names: ['S', 'E1', 'E2', 'E3', 'I', 'R', 'N_EI', 'N_IR']
N_EI is at index: 6
```

# Load Data and Build Model III

```
def rinit(theta_, key, covars, t0=None):
    """Initial state distribution."""
    N = theta_["N"]
    S_0 = theta_["S_0"]
    E_0 = theta_["E_0"]
    I_0 = theta_["I_0"]
    R_0 = theta_["R_0"]

    m = N / (S_0 + E_0 + I_0 + R_0)
    S = jnp.rint(m * S_0)
    E_each = jnp.rint(m * E_0 / nstageE)
    I = jnp.rint(m * I_0)
    R = jnp.rint(m * R_0)

    result = {"S": S, "I": I, "R": R, "N_EI": 0.0, "N_IR": 0.0}
    for i in range(nstageE):
        result[f"E{i+1}"] = E_each

    return result
```

# Load Data and Build Model IV

```
def rproc(X_, theta_, key, covars, t=None, dt=None):
    """Process model: SEIR with Erlang-distributed exposed period."""
    N = theta_["N"]
    R0 = theta_["R0"]
    alpha = theta_["alpha"]
    gamma = theta_["gamma"]

    S = X_["S"]
    E = jnp.array([X_[f"E{i+1}"] for i in range(nstageE)])
    I = X_["I"]
    R = X_["R"]
    N_EI_prev = X_["N_EI"]
    N_IR_prev = X_["N_IR"]

    # Transmission rate
    beta = R0 * gamma
    lam = beta * I / N

    # Transition probabilities (clipped to valid range)
    pS = jnp.clip(1.0 - jnp.exp(-lam * timestep), 0.0, 1.0)
    pE = jnp.clip(1.0 - jnp.exp(-nstageE * alpha * timestep), 0.0, 1.0)
    pI = jnp.clip(1.0 - jnp.exp(-gamma * timestep), 0.0, 1.0)

    # Split keys for binomial draws
```

# Exercise 1: The Sierra Leone Outbreak I

Apply probes to investigate the extent to which the SEIR model above is an adequate description of the data from the Sierra Leone outbreak.

**Tasks:**

1. Have a look at the probes provided (growth rate, residual SD, autocorrelation).
2. Try also to come up with some informative probes of your own.
3. Discuss the implications of your findings.

# Defining Probe Functions I

# Defining Probe Functions II

```python
def growth_rate_probe(y):
    """Estimate exponential growth rate from case data."""
    y = np.array(y).flatten()
    y_safe = np.maximum(y, 0.5)
    log_y = np.log(y_safe)

    t = np.arange(len(y))
    mask = ~np.isnan(log_y) & ~np.isinf(log_y)
    if mask.sum() < 2:
        return np.nan

    t_m = t[mask]
    y_m = log_y[mask]
    slope = np.cov(t_m, y_m)[0, 1] / np.var(t_m)
    return slope

def residual_sd_probe(y):
    """Compute SD of residuals from exponential trend."""
    y = np.array(y).flatten()
    y_safe = np.maximum(y, 0.5)
    log_y = np.log(y_safe)

    t = np.arange(len(y))
    mask = ~np.isnan(log_y) & ~np.isinf(log_y)
```

# Simulating from the Model I

```
# Generate simulations
key = jax.random.key(42)
X_sims, Y_sims = sle.simulate(key=key, nsim=500)

# Get Sierra Leone data
sle_cases = dat_s['cases'].values

print(f"Generated {500} simulations")
print(f"Data length: {len(sle_cases)} weeks")
print(f"\nX_sims columns: {X_sims.columns.tolist()}")
print(f"Y_sims columns: {Y_sims.columns.tolist()}")
```

```
Generated 500 simulations
Data length: 17 weeks

X_sims columns: ['replicate', 'sim', 'time', 'state_0', 'state
Y_sims columns: ['replicate', 'sim', 'time', 'obs_0']
```

# Applying Probes I

```
# Compute probe values for data
data_probes = {
    'growth_rate': growth_rate_probe(sle_cases),
    'residual_sd': residual_sd_probe(sle_cases),
    'acf_1': acf_probe(sle_cases, lag=1),
    'acf_2': acf_probe(sle_cases, lag=2),
    'max_cases': max_cases_probe(sle_cases),
    'total_cases': total_cases_probe(sle_cases),
    'peak_week': peak_week_probe(sle_cases)
}

print("Data probe values:")
for name, val in data_probes.items():
    print(f"  {name}: {val:.4f}")
```

# Applying Probes II

```
Data probe values:
  growth_rate: 0.1612
  residual_sd: 0.4982
  acf_1: 0.6947
  acf_2: 0.4923
  max_cases: 299.5460
  total_cases: 2085.2640
  peak_week: 16.0000
```

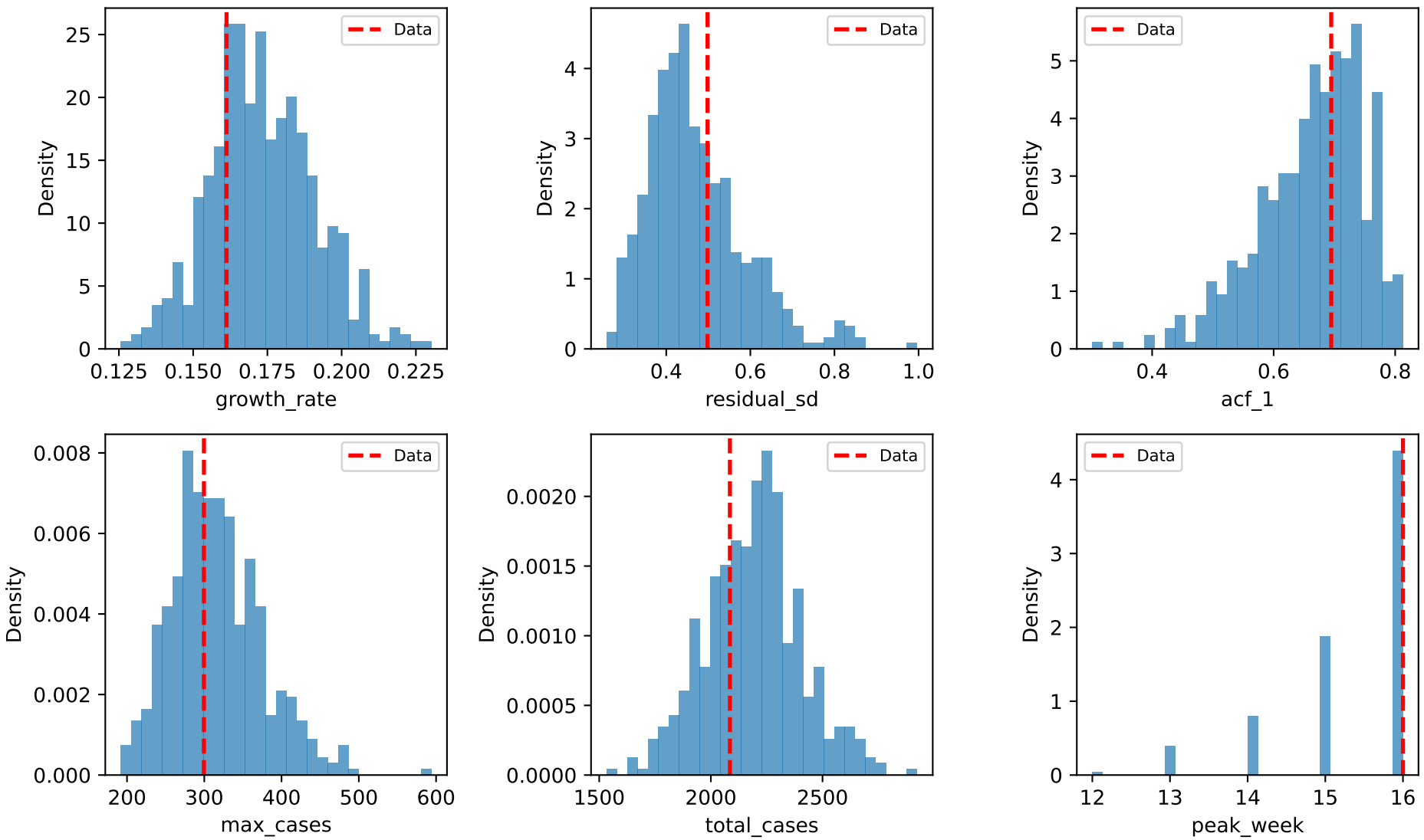
# Applying Probes III

```python
# Compute probe values for simulations
sim_probes = {name: [] for name in data_probes.keys()}

for i in range(500):
    sim_data = Y_sims[Y_sims['sim'] == i]
    cases = sim_data['obs_0'].values

    sim_probes['growth_rate'].append(growth_rate_probe(cases))
    sim_probes['residual_sd'].append(residual_sd_probe(cases))
    sim_probes['acf_1'].append(acf_probe(cases, lag=1))
    sim_probes['acf_2'].append(acf_probe(cases, lag=2))
    sim_probes['max_cases'].append(max_cases_probe(cases))
    sim_probes['total_cases'].append(total_cases_probe(cases))
    sim_probes['peak_week'].append(peak_week_probe(cases))

# Convert to arrays
for name in sim_probes:
    sim_probes[name] = np.array(sim_probes[name])
```

# Visualizing Probe Results I

```
# Plot probe distributions
fig, axes = plt.subplots(2, 3, figsize=(10, 6))
axes = axes.flatten()

probe_names = ['growth_rate', 'residual_sd', 'acf_1',
               'max_cases', 'total_cases', 'peak_week']

for ax, name in zip(axes, probe_names):
    sim_vals = sim_probes[name]
    sim_vals = sim_vals[~np.isnan(sim_vals)]

    ax.hist(sim_vals, bins=30, alpha=0.7, density=True)
    ax.axvline(data_probes[name], color='red', linewidth=2,
               linestyle='--', label='Data')
    ax.set_xlabel(name)
    ax.set_ylabel('Density')
    ax.legend(fontsize=8)

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

# Visualizing Probe Results II



# Visualizing Probe Results III

```
# Compute p-values
print("Probe p-values (two-sided):")
for name in probe_names:
    sim_vals = sim_probes[name]
    sim_vals = sim_vals[~np.isnan(sim_vals)]
    data_val = data_probes[name]

    # Two-sided p-value
    p_lower = np.mean(sim_vals <= data_val)
    p_upper = np.mean(sim_vals >= data_val)
    p_val = 2 * min(p_lower, p_upper)

    print(f"  {name}: p = {p_val:.4f}")
```

# Visualizing Probe Results IV

```
Probe p-values (two-sided):
  growth_rate: p = 0.4720
  residual_sd: p = 0.6680
  acf_1: p = 0.8400
  max_cases: p = 0.8480
  total_cases: p = 0.5960
  peak_week: p = 1.1720
```

# Interpretation I

```
print("Interpretation of probe results:")
print("")
print("1. Growth Rate:")
print("   If p-value is extreme, the model may not capture")
print("   the actual growth dynamics well.")
print("")
print("2. Residual SD:")
print("   If data has lower SD than simulations, the model")
print("   may be adding too much stochastic variation.")
print("")
print("3. Autocorrelation:")
print("   Mismatch suggests the model's temporal dynamics")
print("   don't match the observed patterns.")
print("")
print("4. Max/Total Cases:")
print("   These indicate whether the model captures the")
print("   overall scale of the outbreak.")
```

# Interpretation II

Interpretation of probe results:

1. Growth Rate:
   If p-value is extreme, the model may not capture the actual growth dynamics well.

2. Residual SD:
   If data has lower SD than simulations, the model may be adding too much stochastic variation.

3. Autocorrelation:
   Mismatch suggests the model's temporal dynamics don't match the observed patterns.

4. Max/Total Cases:

# Interpretation III

These indicate whether the model captures the overall scale of the outbreak.

# Exercise 2: Decomposing the Uncertainty I

As we have discussed, the uncertainty shown in the forecasts above has three sources:

1. **Measurement error**: Uncertainty in reported cases given true incidence
2. **Process noise**: Stochasticity in disease transmission
3. **Parametric uncertainty**: Uncertainty in parameter estimates

Show how you can break the total uncertainty into these three components. Produce plots similar to that above showing each of the components.

# Strategy for Decomposition I

To decompose uncertainty:

1. **Measurement error only**: Fix parameters at MLE, use deterministic skeleton, vary only observation process
2. **Process noise only**: Fix parameters at MLE, run stochastic simulations, use expected observations
3. **Parameter uncertainty only**: Sample parameters, use deterministic trajectories with expected observations
4. **Total uncertainty**: Sample parameters, run stochastic simulations, vary observations

# Strategy for Decomposition II

```python
# Get MLE parameters
theta_mle = sle.theta[0]
rho = theta_mle["rho"]
k = theta_mle["k"]
times = dat_s['week'].values
n_times = len(times)

print(f"MLE rho: {rho:.4f}")
print(f"MLE k: {k:.4f}")
print(f"Number of time points: {n_times}")
```

```
MLE rho: 0.2000
MLE k: 0.0385
Number of time points: 17
```

# Strategy for Decomposition III

```
# Define deterministic skeleton (for comparison)
def deterministic_trajectory(theta_, times, pop):
    """Compute deterministic SEIR trajectory."""
    N = pop
    R0 = theta_["R0"]
    alpha = theta_["alpha"]
    gamma = theta_["gamma"]

    beta = R0 * gamma
    m_alpha = nstageE * alpha

    # Initial conditions (using same logic as rinit)
    total = theta_["S_0"] + theta_["E_0"] + theta_["I_0"] + theta_["R_0"]
    S = N * theta_["S_0"] / total
    E_total = N * theta_["E_0"] / total
    E1 = E_total / 3
    E2 = E_total / 3
    E3 = E_total / 3
    I = N * theta_["I_0"] / total
    R = N * theta_["R_0"] / total

    dt = 0.1
    results = []
```

# Component 1: Measurement Error Only I

```
# Get deterministic trajectory at MLE
det_traj = deterministic_trajectory(theta_mle, times,
                                    populations["SierraLeone"])

# Sample observations from deterministic incidence
n_sims = 100
meas_error_sims = []

for i in range(n_sims):
    # True incidence from deterministic model
    H = det_traj['N_EI'].values
    mu = rho * H

    # Sample observations (negative binomial)
    # Using numpy's parameterization: n=k, p=k/(k+mu)
    cases = np.random.negative_binomial(k, k/(k + mu + 1e-10))
    meas_error_sims.append(cases)

meas_error_sims = np.array(meas_error_sims)
print(f"Measurement error simulations: {meas_error_sims.shape}")
```

# Component 1: Measurement Error Only II

```
Measurement error simulations: (100, 17)
```

# Component 2: Process Noise Only I

# Component 2: Process Noise Only II

```
# Run stochastic simulations at MLE, report expected observations
# (mean of negative binomial, no sampling)

key = jax.random.key(12345)
X_sims_proc, Y_sims_proc = sle.simulate(key=key, nsim=100)

# Check available columns
print(f"X_sims_proc columns: {X_sims_proc.columns.tolist()}")

# Find the column for N_EI (it might be named differently)
n_ei_idx = statenames.index('N_EI')
print(f"N_EI index in statenames: {n_ei_idx}")

# Try to find the right column name
if 'N_EI' in X_sims_proc.columns:
    n_ei_col = 'N_EI'
elif f'state_{n_ei_idx}' in X_sims_proc.columns:
    n_ei_col = f'state_{n_ei_idx}'
else:
    # List all state-like columns
    state_cols = [c for c in X_sims_proc.columns if c not in ['time', 'sim']]
    n_ei_col = state_cols[n_ei_idx] if len(state_cols) > n_ei_idx else state_cols[-1

print(f"Using column for N_EI: {n_ei_col}")
```

# Component 3: Parameter Uncertainty Only I

# Component 3: Parameter Uncertainty Only II

```python
# Sample parameters, compute deterministic trajectories
def sample_params(mle, n=100, scale=0.15):
    params_list = []
    for _ in range(n):
        params = mle.copy()
        params["R0"] = mle["R0"] * np.exp(np.random.normal(0, scale))
        params["rho"] = mle["rho"] * np.exp(np.random.normal(0, scale))
        params["k"] = mle["k"] * np.exp(np.random.normal(0, scale))
        params_list.append(params)
    return params_list

np.random.seed(42)
param_samples = sample_params(theta_mle, n=100)

param_uncert_sims = []
for params in param_samples:
    traj = deterministic_trajectory(params, times,
                                    populations["SierraLeone"])
    expected_cases = params["rho"] * traj['N_EI'].values
    param_uncert_sims.append(expected_cases[:n_times])

param_uncert_sims = np.array(param_uncert_sims)
print(f"Parameter uncertainty simulations: {param_uncert_sims.shape}")
```

# Comparing Uncertainty Components I

# Comparing Uncertainty Components II

```
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# Plot 1: Measurement error only
ax = axes[0, 0]
for i in range(min(20, n_sims)):
    ax.plot(times, meas_error_sims[i][:n_times], alpha=0.2, color='blue')
ax.plot(dat_s['week'], dat_s['cases'], 'k-', linewidth=2)
ax.set_title('Measurement Error Only')
ax.set_xlabel('Week')
ax.set_ylabel('Cases')

# Plot 2: Process noise only
ax = axes[0, 1]
for i in range(min(20, 100)):
    sim_vals = process_noise_sims[i]
    ax.plot(times[:len(sim_vals)], sim_vals, alpha=0.2, color='green')
ax.plot(dat_s['week'], dat_s['cases'], 'k-', linewidth=2)
ax.set_title('Process Noise Only')
ax.set_xlabel('Week')
ax.set_ylabel('Expected Cases')

# Plot 3: Parameter uncertainty only
ax = axes[1, 0]
for i in range(min(20, 100)):
```

# Quantifying Uncertainty Components I

# Quantifying Uncertainty Components II

```python
# Compute variance at each time point for each component
def compute_variance_over_time(sims):
    """Compute variance at each time point."""
    return np.var(sims, axis=0)

# Ensure all arrays have the same length
min_len = min(meas_error_sims.shape[1],
              process_noise_sims.shape[1],
              param_uncert_sims.shape[1],
              n_times)

var_meas = compute_variance_over_time(meas_error_sims[:, :min_len])
var_process = compute_variance_over_time(process_noise_sims[:, :min_len])
var_param = compute_variance_over_time(param_uncert_sims[:, :min_len])

# Total variance from full simulations
full_sims = []
for i in range(100):
    sim_data = Y_sims_proc[Y_sims_proc['sim'] == i]
    sim_cases = sim_data['obs_0'].values[:min_len]
    full_sims.append(sim_cases)
full_sims = np.array(full_sims)
var_total = compute_variance_over_time(full_sims)
```

# Interpretation I

```
print("Interpretation of Uncertainty Decomposition:")
print("")
print("1. MEASUREMENT ERROR:")
print("   - Arises from imperfect observation of true incidence")
print("   - Typically largest at peak (more cases to miscount)")
print("   - Can be reduced with better surveillance")
print("")
print("2. PROCESS NOISE:")
print("   - Inherent stochasticity in disease transmission")
print("   - Important early in outbreak (small numbers)")
print("   - Cannot be eliminated")
print("")
print("3. PARAMETER UNCERTAINTY:")
print("   - Uncertainty in R0, rho, k, etc.")
print("   - Grows over time (trajectories diverge)")
print("   - Can be reduced with more data")
```

# Interpretation II

Interpretation of Uncertainty Decomposition:

1. MEASUREMENT ERROR:
   - Arises from imperfect observation of true incidence
   - Typically largest at peak (more cases to miscount)
   - Can be reduced with better surveillance

2. PROCESS NOISE:
   - Inherent stochasticity in disease transmission
   - Important early in outbreak (small numbers)
   - Cannot be eliminated

3. PARAMETER UNCERTAINTY:
   - Uncertainty in R0, rho, k, etc.
   - Grows over time (trajectories diverge)

# Interpretation III

- Can be reduced with more data

# Summary I

**Exercise 1 (Sierra Leone Probes):**

- Multiple probes reveal different aspects of model fit
- Growth rate and residual SD are particularly informative
- Autocorrelation probes capture temporal dynamics
- P-values quantify discrepancy between model and data

# Summary II

**Exercise 2 (Uncertainty Decomposition):**

- Three main sources: measurement, process, parameter
- Relative importance varies over time
- Early: process noise dominates (small numbers)
- Peak: measurement error dominates
- Late: parameter uncertainty dominates (trajectory divergence)
- Understanding sources guides intervention/data collection

# Acknowledgments I


- This lesson is prepared for the Simulation-based Inference for Epidemiological Dynamics module at SISMID.
- The materials build on previous versions of this course and related courses.

# References