Extensions to the pomp package and framework

Spencer J. Fox   Qianying (Ruby) Lin   Jesse Wheeler

Package Extensions

Mathematically, POMP models are very versatile, and many of the ideas useful to the models described so far are also applicable in more general settings. This gives rise to a few useful extensions of the pomp package, which we try to describe here.
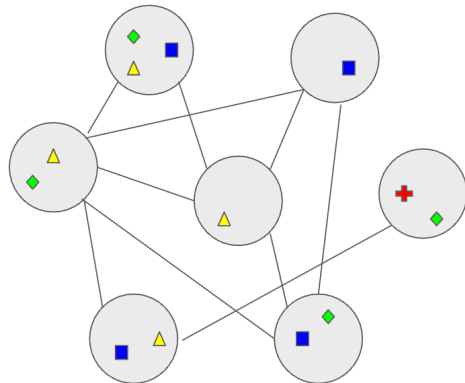
- ▶ Meta-population data and models
  - ▶ `panelPomp`
  - ▶ `spatPomp`
- ▶ Genomic / phylogenetic data
  - ▶ `phylopomp`

Each of these packages extend the pomp package to other useful cases.

Meta-population: Data and Models
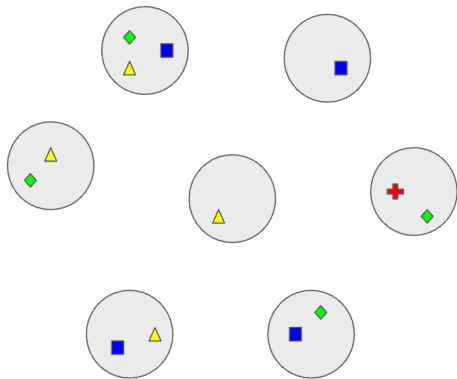
panelPomp: data from related, independent systems.

spatPomp: data from a single interacting system.

Meta-population packages: Examples

Both packages require `pomp` to be installed, and the internal workings of both packages actually use many of the `pomp` functions. Thus, familiarity with `pomp` is a prerequisite.

▶ `panelPomp`:
  - ▶ We have *panel* or *longitudinal* data.
  - ▶ Several independent yet related systems. Information from all systems is useful, not just one location.
  - ▶ Simultaneously allows for features shared by each system as well as system unique features.
  - ▶ Example: endemic COVID-19 measured in 3 locations: New York, London, Atlanta.

▶ `spatPomp`:
  - ▶ The name comes from the idea that we have spatially explicit POMP models.
  - ▶ The data at multiple locations are thought of as location-specific measurements from the same system. That is, the dynamic systems underlying the data are connected.
  - ▶ Example: Modeling early stages of COVID-19 in China. Here, we might have measurements for many cities, but at first cases were only found in Wuhan.

PanelPomp: a collection of POMP models

A `panelPomp` model is really just a collection of `pomp` models. This is also how they are built:

```r
library(panelPomp)

mod1 <- pomp(..., params = c('p1' = 0.1, 'p2' = 1.2, 'p3' = 0.9))
mod2 <- pomp(..., params = c('p1' = 0.1, 'p2' = 1.1, 'p3' = 0.6))
mod3 <- pomp(..., params = c('p1' = 0.1, 'p2' = 1.5, 'p3' = 0.75))

ppomp <- panelPomp(
  object = list(mod1, mod2, mod3),
  shared = c("p1" = 0.1), # Shared-value parameters
  specific = c("p2", "p3") # Unit-Specific parameters
)
```
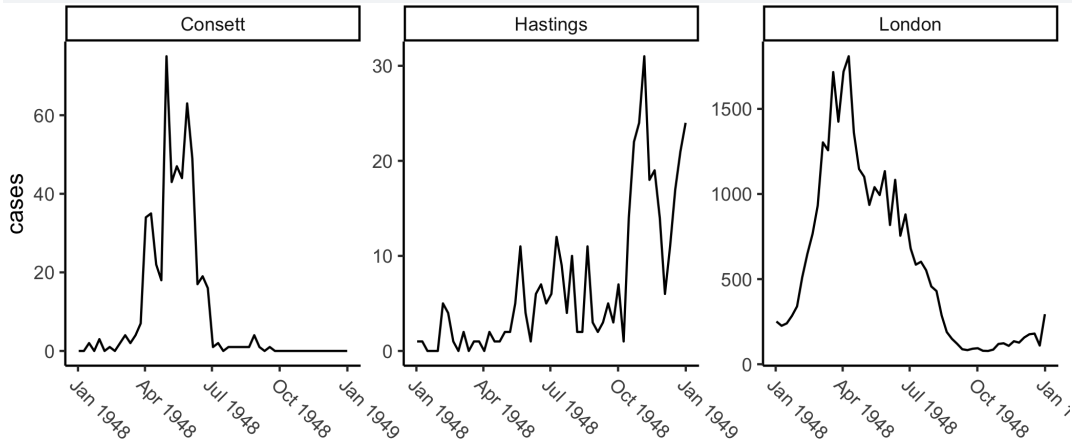
Measles Example

We can build off of the measles example by looking at UK measles from multiple cities, and building an SEIR model for the data. There is built in models and data in `panelPomp`

```
measSIR <- panelMeasles(
  units = c("Consett", "London", "Hastings"),
  first_year = 1948,
  last_year = 1948
  )
```

Measles Example: Figure

```
plot(measSIR)
```

panelPomp: parameters

Parameter names in `panelPomp` have the following conventions. If the parameter is shared, it just is called by the name. If it is unit-specific, the name of the unit follows the name of the parameter: `<param>[<unit>]`.

```
coef(measSIR)
```

mu	sigma[Consett]	gamma[Consett]
2.00e-02	4.26e+01	1.72e+02
rho[Consett]	R0[Consett]	amplitude[Consett]
6.50e-01	3.59e+01	2.00e-01
alpha[Consett]	iota[Consett]	cohort[Consett]
1.01e+00	7.31e-02	3.10e-01
psi[Consett]	S_0[Consett]	E_0[Consett]
4.06e-01	3.22e-02	1.83e-05
I_0[Consett]	R_0[Consett]	sigmaSE[Consett]
1.97e-05	9.68e-01	7.12e-02
sigma[Hastings]	gamma[Hastings]	rho[Hastings]
5.63e+01	7.41e+01	6.95e-01

panelPomp: shared vs unit-specific

A key feature of `panelPomp` objects is which parameters are shared, which are unit-specific. If parameters are shared, that means they have the same value for all units. We can inspect and modify which parameters are which using the functions `shared` and `specific`:

```
shared(measSIR)
```

```
   mu
0.02
```

```
shared(measSIR) <- c(shared(measSIR), 'alpha' = 1)
shared(measSIR)
```
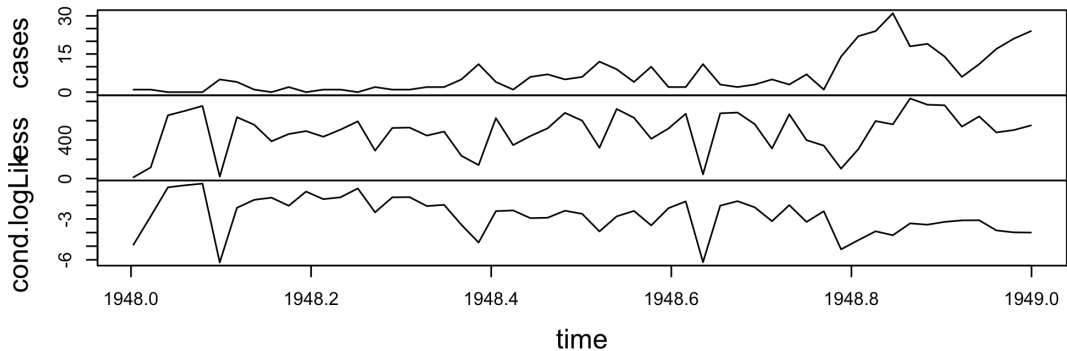
```
   mu alpha
 0.02  1.00
```

panelPomp Functions

panelPomp is the easiest extension: If you can build one pomp, you can build multiple into a single panelPomp object. Existing functions and algorithms are similar as well!

```
pfilter(measSIR, Np = 1000) |> plot(unit = 'Hastings')
```


Hastings

panelPomp: iterated filtering

We need a new algorithm for conducting maximum likelihood, either the panel iterated filter (PIF), or marginalized panel iterated filter (MPIF). We don't even need to change the code!

```
mif2Out <- mif2(
  measSIR, rw.sd = rw_sd(rho = 0.02, R_0 = 0.02),
  Nmif = 10, Np = 200, cooling.fraction.50 = 0.5,
  block = TRUE  # block = TRUE does MPIF, usually best + faster.
  )
```

panelPomp: likelihood I

Because units are independent, the likelihood can be computed independently for each unit, and the total log-likelihood is the sum of unit log-likelihoods.

```
unitLogLik(mif2Out)
```

```
  Consett  Hastings    London
-112.5131 -138.7637 -400.0030
```

```
sum(unitLogLik(mif2Out))
```

```
[1] -651.2799
```

```
logLik(mif2Out)
```

```
[1] -651.2799
```

panelPomp: likelihood II

When we do repeated particle filters, we get better estimates if we first average log-likelihood over the units, then take the sum.

```
library(foreach)
library(doParallel)
library(doRNG)
registerDoParallel(detectCores()-1)
foreach(rep=1:10, .combine = c) %dopar% {
  pfilter(measSIR, Np = 5000)
} -> pfOut  # Object of length 10
```

panelPomp: likelihood III

Like in pomp, we can average results using `logmeanexp`. However, because there is more than one way to do it, we have a new function `panel_logmeanexp`. The preferred approach is averaging over the unit, then summing:

```
panel_logmeanexp(
  sapply(pfOut, unitLogLik), MARGIN = 1, se = TRUE
)
```

```
                    se
-701.877291    3.339815
```

We usually have larger error than in pomp, because there is error for each unit that adds up across the entire `panelPomp` object.

panelPomp: sumary

▶ Easy extension of `pomp` that allows to fit related models to many locations.
▶ Useful for (approximately) independent systems (e.g., outbreaks within various hospitals, approximately geographically isolated outbreaks of same disease, controlled randomized experiments,…)
▶ Very useful if limited data is available, we can pool information from all sources.
▶ Allows for estimating shared and unit-specific parameters.
▶ To learn more, see the panelPomp research paper (to appear in the R Journal) (Bretó et al. 2025), or the package website https://panelpomp-org.github.io/

panelPomp: Exercise

Install the `panelPomp` package from CRAN via `install.packages("panelPomp")`.
Create one of the prebuilt `panelPomp` models using a constructor function:
`panelGompertz()`, `panelMeasles()`, `contacts`, or `panelRandomWalk()`. Our goal
is to build some familiarity with `panelPomp` models.

Check the documentation to learn more about the model (e.g., `?panelGompertz`).
How many units does it have by default? How many observations and parameters?
Which parameters are shared and which unit specific? Can you change the parameter
type? What happens if you `plot` the object?

Once you feel like you understand the object, try running a particle filter and iterated
filter (PIF or MPIF, implemented via `mif2`).

spatPomp: Introduction

spatPomp is another extension that can be used to fit meta-population models. The difference between spatPomp and panelPomp is that spatPomp allows for dependence between units.

▶ Both mathematically and computationally, this is a much more difficult task.
▶ It can lead to very interesting and high-dimensional models.

Note that any panelPomp model can be expressed as a spatPomp, so you can use spatPomp methods to fit panelPomp models. However, if a panelPomp is appropriate, it is both theoretically and computationally more efficient to treat the model as a panelPomp.

High-dimensional models

spatPomp is relatively new, the software paper was published last year (Asfaw et al. 2024b). It enables fitting higher-dimensional POMP models than has been possible in the past.
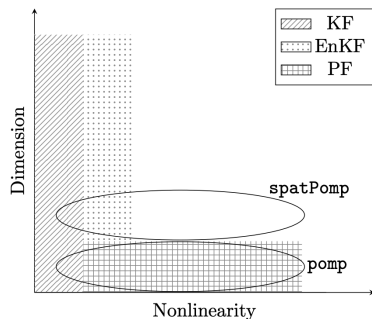

Figure 1: Limits of POMP modeling. Credit: Asfaw et al. 2024.

Dependent States

Unlike panelPomp, the latent process $X(t)$ is assumed to be linked for each unit (e.g., infectious individuals from city 1 can infect susceptible individuals in city 2).
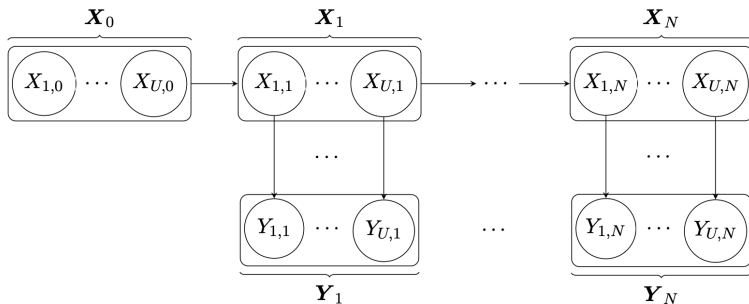

Figure 2: Credit: Asfaw et al. 2024.

Implications

▶ Allowing for spatially dependent latent processes can lead to more interesting, more realistic models.

▶ However, they become more complicated to build in practice, because we have to carefully describe how the systems interact.

▶ Mathematically, these models are also more complex. Until very recently, plug-and-play methods for fitting and evaluating these models did not exist.

spatPomp: methods

Parts of a `spatPomp` that can be implemented

Method	Argument to `spatPomp()`	Mathematical terminology
dunit_measure	dunit_measure	Evaluate $f_{Y_{u,n}\mid X_{u,n}}(y_{u,n}\mid x_{u,n};\theta)$
runit_measure	runit_measure	Simulate from $f_{Y_{u,n}\mid X_{u,n}}(y_{u,n}\mid x_{u,n};\theta)$
eunit_measure	eunit_measure	Evaluate $e_{u,n}(x,\theta)=E[Y_{u,n}\mid X_{u,n}=x;\theta]$
vunit_measure	vunit_measure	Evaluate $v_{u,n}(x,\theta)=\mathrm{Var}[Y_{u,n}\mid X_{u,n}=x;\theta]$
munit_measure	munit_measure	$m_{u,n}(x,V,\theta)=\psi$ if $v_{u,n}(x,\psi)=V$, $e_{u,n}(x,\psi)=e_{u,n}(x,\theta)$
rprocess	rprocess	Simulate from $f_{\boldsymbol{X}_n\mid\boldsymbol{X}_{n-1}}(\boldsymbol{x}_n\mid\boldsymbol{x}_{n-1};\theta)$
dprocess	dprocess	Evaluate $f_{\boldsymbol{X}_n\mid\boldsymbol{X}_{n-1}}(\boldsymbol{x}_n\mid\boldsymbol{x}_{n-1};\theta)$
rmeasure	rmeasure	Simulate from $f_{\boldsymbol{Y}_n\mid\boldsymbol{X}_n}(\boldsymbol{y}_n\mid\boldsymbol{x}_n;\theta)$
dmeasure	dmeasure	Evaluate $f_{\boldsymbol{Y}_n\mid\boldsymbol{X}_n}(\boldsymbol{y}_n\mid\boldsymbol{x}_n;\theta)$
rprior	rprior	Simulate from the prior distribution $\pi(\theta)$
dprior	dprior	Evaluate the prior density $\pi(\theta)$
rinit	rinit	Simulate from $f_{\boldsymbol{X}_0}(\boldsymbol{x}_0;\theta)$
timezero	t0	t_0
time	times	$t_{1:N}$
obs	data	$\boldsymbol{y}^*_{1:N}$
states	—	$\boldsymbol{x}_{0:N}$
coef	params	θ

Block particle filter (`bpfilter`)

To perform inference, it is similar to pomp, but we will use an algorithm called the Block-Particle Filter (BPF). Particle filters don't work well in high-dimensions (even as low as 3 dimensions is "high"). The BPF is similar to the particle filter, but does the following:

▶ Simulate the entire process forward with `rprocess`.
▶ Use `dunit_meas` to calculate weights independently for all units, and update particles independently.

Pro: This algorithm is surprisingly fast and effective. Best if coupling between units is weak.

Con: Does not satisfy conservation of mass properties. Alternative options can be found in the `spatPomp` tutorial (Asfaw et al. 2024a).

Iterated block particle filter (IBPF)

Like iterating filtering is a special type of particle filter, with model parameters doing a random walk, we can iterate the BPF to get an IBPF to fit model parameters (Ionides, Ning, and Wheeler 2024).

Like the BPF, the IBPF is surprisingly effective at maximizing the likelihood (Asfaw et al. 2024a), but does not conserve mass.

Example: Cholera in Haiti

- Haiti experienced a cholera outbreak following the devastating 2010 earthquake.
- From 2010-2019, more than 800,000 recorded cases, making it one of the largest recorded outbreaks.
- Oral cholera vaccination (OCV) is available, but in limited supply.
- A modeling goal is to study the potential impact of various vaccination strategies.
- Image credit: UNICEF.
- This example is from Wheeler et al. 2024.

Haiti Data
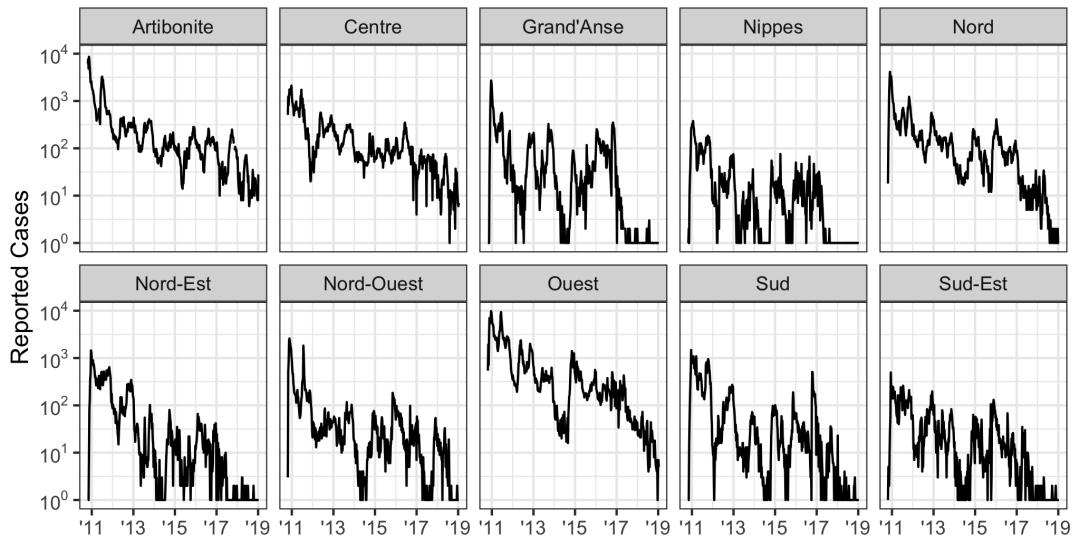
Cholera Model
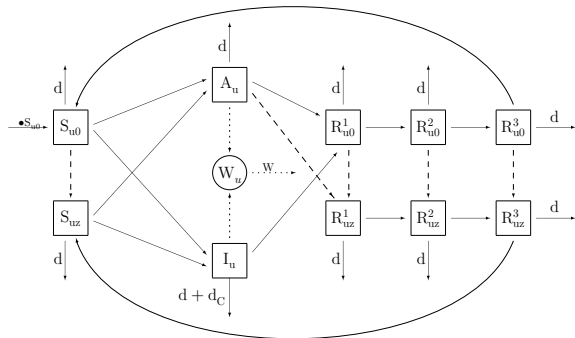
- ▶ SIAR3W Model
- ▶ Spatial Dependence between units.
- ▶ Stochastic transmission rates.
- ▶ Overdispersed Markov counting system.
- ▶ Rainfall driven transmission.
- ▶ Environmental reservoir of bacteria.
- ▶ Adjustments for Hurricane Mathew (Oct 2016).
- ▶ For details on implementation, see (Wheeler et al. 2024).

Pre-built model in the `haitipkg` R package

Rather than build this from scratch, we can use a constructor function from `haitipkg`.

```
devtools::install_github('jeswheel/haitipkg')
sp_mod <- haitipkg::haiti3_spatPomp()
```

Performing inference for this model is similar to the `pomp` Measles example, but it takes much longer to compute in practice

```
bpf_out <- bpfilter(sp_mod, Np = 1000, block_size = 1)
```

This took 1.97 seconds to run.

Parameter Estimation

The functionality for estimating parameters using IBPF is similar to using `mif2` in pomp:

```
## NOT RUN
ibpf(
  h3_spat,
  Nbpf = 100,  # Equivalent to Nmif
  Np = 2000,
  sharedParNames = <character vector of shared parameter names>,
  unitParNames = <character vector of unit-specific parameter names>,
  spat_regression = 0.1,
  rw.sd = ...,
  cooling.fraction.50 = ...,
  block_size = 1
)
```

Confidence Intervals

For `spatPomp`, we can compute confidence intervals in a similar way as `pomp`. That is, we can fix the parameter of interest at a range of values, and maximize the likelihood for the remaining parameters.
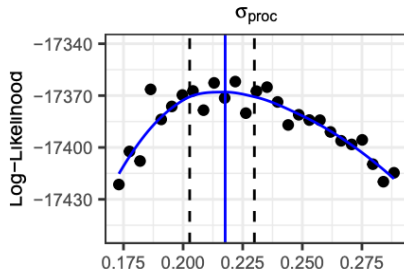
Figure 3: Confidence interval for one of the model parameters. Credit: Wheeler et al. 2024.

spatPomp: Summary

▶ The spatPomp package is an extension of pomp that enables building and fitting interacting dynamic meta-population models.

▶ Algorithms and software are a little bit different than pomp, but similar enough that you pomp skills are applicable.

▶ The nature of these models mean that things run much slower, for two reasons:
   ▶ If you have data from U units, then the code is roughly U times slower for the same number of iterations and particles.
   ▶ Because there are U times as many parameters, the likelihood surface is usually more complex, and we may need more iterations and particles than for low-dimensional systems.

▶ To learn more, see the package tutorial (Asfaw et al. 2024a) or the Haiti Cholera example we discussed (Wheeler et al. 2024).
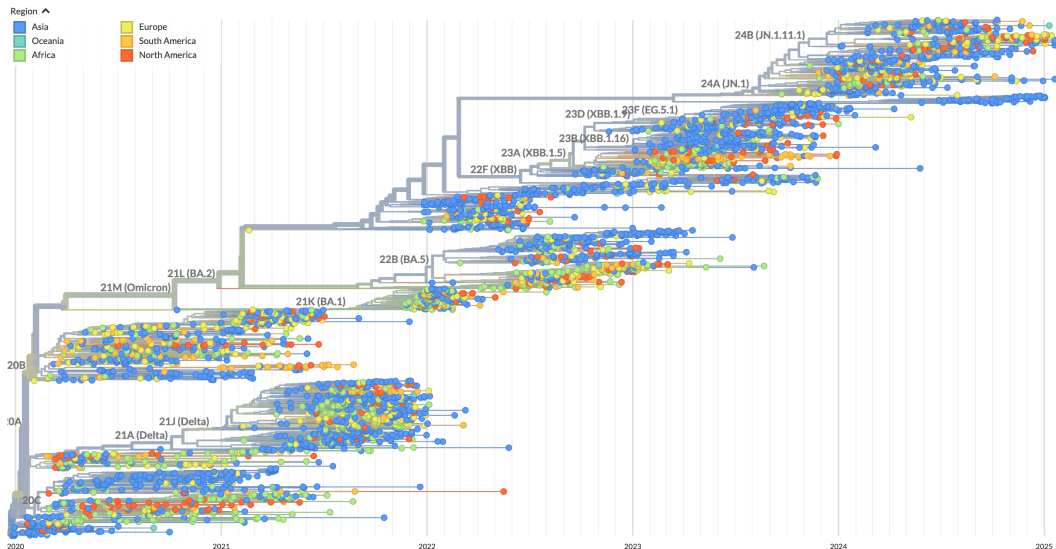
Phylodynamic package: phylopomp

The spreading of infectious diseases can be considered as the mutation accumulation in genomic sequences in hosts and the transmission of such virus from host to host. Therefore, when the viral genomic samples are collected from hosts, one can traceback a partially observed ancestral history and reconstruct it as a tree, called genealogy or phylogeny.

With `phylopomp`, we can simulate these partially observed ancestral history from a board class of epidemiological models, and infer the transmission dynamics given a genealogy/phylogeny based on `pomp`.

The mathematical theories and algorithm are discussed in King, Lin, and Ionides (2025).

phylopomp: genomic sequences & phylogenetic tree
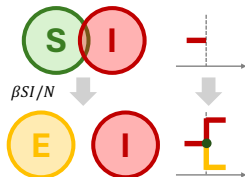
phylopomp: phylogenetics & phylodynamics
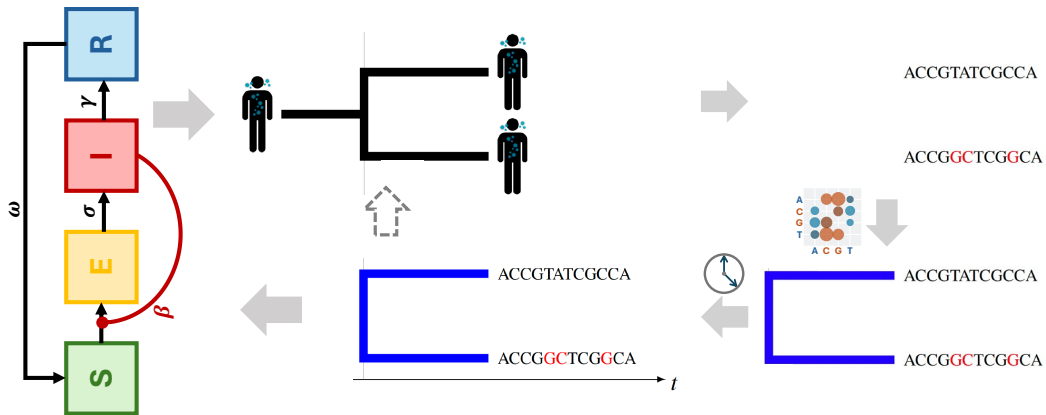
The goal of `phylopomp` is to infer the partially observed epidemiological dynamics, formulated by compartmental models, using the genealogy/phylogeny, reconstructed from sampled genomic sequences.
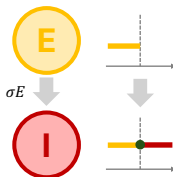
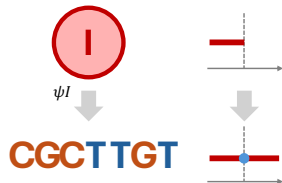# phylopomp: genealogy and events

A: birth-type
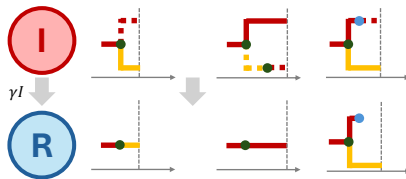
$\beta SI/N$

transmission v. branching

B: migration-type

σE

progression v. transition

C: sample-type

ψI

CGCTTGT

sampling

D: death-type

γI

recovery v. death

phylopomp: installation

The github repo for phylopomp is on https://github.com/kingaa/phylopomp/.
We can install the latest released version from github using devtools.

```
library(devtools)
install_github("kingaa/phylopomp@0.14.8.0")
```

To confirm the package is installed successfully,

```
packageVersion("phylopomp")
```

```
[1] '0.14.8.0'
```

```
library(phylopomp)
```

phylopomp: simulation and inference

▶ The first useful function of phylopomp is to build customized models and simulate genealogies from it. Pre-defined models are included: linear birth-death model (lbdp), moran model, SIR, SEIR, two-class SIR model with super-infection (si2r), two-strain SIR model (siir), etc.

▶ The second is to infer the models given genealogies using pomp. This function is currently under development, while a few models are available: lbdp, moran, sir, and seir.

Example: simulation with pre-defined models in phylopomp I

```r
set.seed(1234)
simulate(
  "SIR",time=2,
  Beta=2,gamma=1,psi=2,
  S0=1000,I0=5
)|>
  # update params
  simulate(
    time=5,
    Beta=5,gamma=2,psi=3
  ) -> model.sir
plot(model.sir)
```
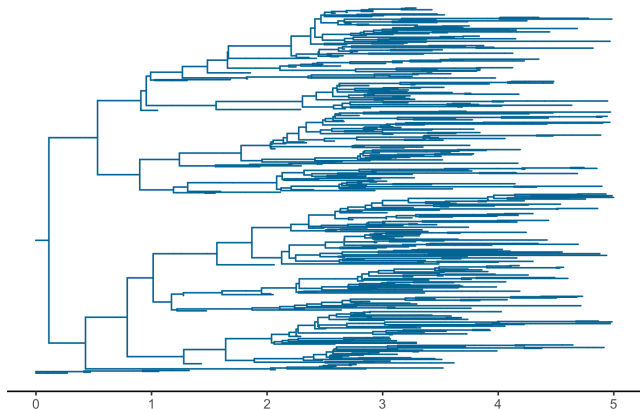

Figure 4: Simulated genealogy from an SIR model.

Example: simulation with pre-defined models in phylopomp II

```
set.seed(1234)
simulate(
  "SEIR",time=2,
  Beta=2,sigma=2,
  gamma=1,psi=2,
  S0=1000,E0=0,I0=5
) |>
  simulate(
    time=5,
    Beta=5,gamma=2,
    psi=3
  ) -> model.seir
plot(model.seir)
```
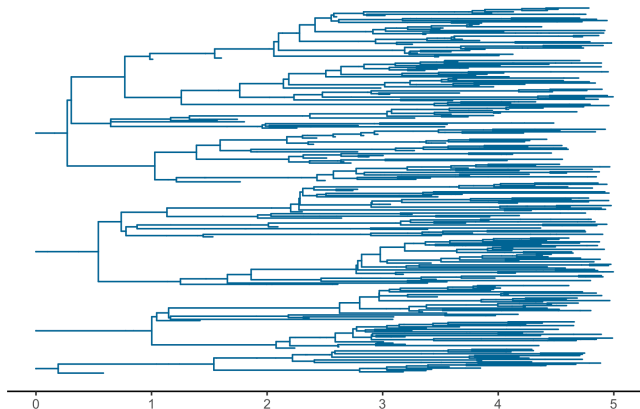

Figure 5: Simulated genealogy from an SEIR model.

Example: simulation with pre-defined models in phylopomp III

```
# show E/I transitions
model.seir |>
  plot(obscure=FALSE)
```

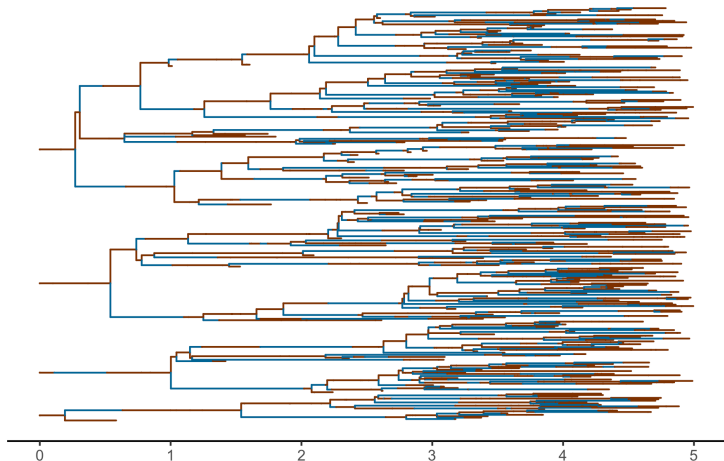The compartments E and I are colored in blue and brown.

Figure 6: Simulated genealogy with colors.

Example: simulation with pre-defined models in phylopomp IV

```
model.seir |>
  lineages(
    obscure=FALSE
  ) |>
  plot()
```
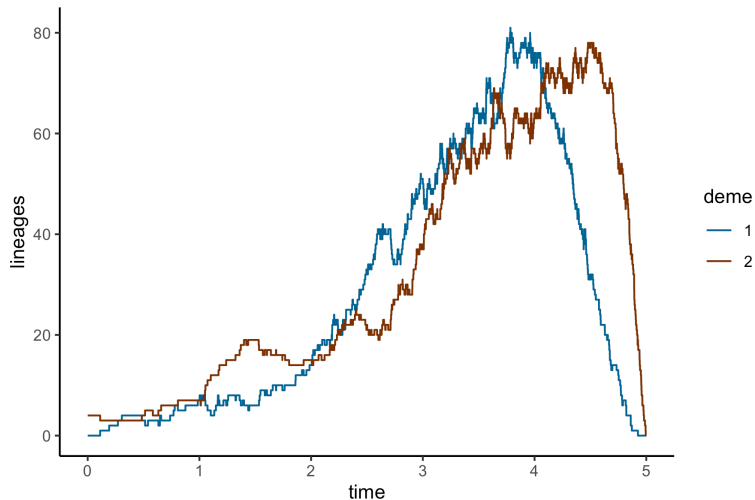

Figure 7: Lineages Through Time (LTT).

Example: inference with pre-defined models in phylopomp I

```
library(pomp)
library(phylopomp)
set.seed(1234)
# simulate a geneal object from an SIR model
simulate("SIR",time=10,
  Beta=3,gamma=1,psi=2,omega=1,S0=100,I0=5) -> x
x |>        # build a pomp object
  sir_pomp(
    Beta=3,gamma=1,psi=2,omega=1,
    S0=100,I0=5,R0=0
  ) -> po
po |> pfilter(Np=5000) -> pf
pf |> logLik()
```

```
[1] -326.5652
```

Example: inference with pre-defined models in phylopomp II

```
set.seed(1234)
simulate("SEIRS",        # simulate a genealogy from an SEIR model
  Beta=4,sigma=1,gamma=1,psi=1,omega=1,
  S0=100,E0=3,I0=5,R0=100, time=5
) -> G
G |>
  seirs_pomp(
    Beta=4,sigma=1,gamma=1,psi=1,omega=1,
    S0=100,E0=3,I0=5,R0=100
  ) |> pfilter(Np=1000) |>
  replicate(n=20) |> concat() -> pf
pf |> logLik() |> logmeanexp(se=TRUE)
```

```
        est           se
-168.856637    1.265223
```

Exercise 1: simulate a genealogy from other pre-defined models I

Run the following code to simulate genealogies from two other pre-defined models and to infer the dynamics from a tree simulated from an SEIR model. Please also see the exercise script `scripts/exercise_phylopomp.r`.

```
library(phylopomp)
# simulate from a two-strain SIR model
??siir                  # check the model setting

simulate(
  "SIIR",Beta1=2,Beta2=50,gamma=1,psi1=2,psi2=1,
  S0=300,I1_0=20,I2_0=2,time=5
) -> model.siir

model.siir |> plot(obscure=FALSE)
model.siir |> lineages(obscure=FALSE) |> plot()
```

Exercise 1: simulate a genealogy from other pre-defined models II

```
# simulate from a two-class SIR model with super-spreader
??si2r                 # check the model setting

simulate(
  "SI2R", Beta=5,mu=2,gamma=1,psi1=1,psi2=2,sigma12=1,sigma21=3,
  S0=500,I0=10,time=5
) -> model.si2r

model.si2r |> plot(obscure=FALSE)
model.si2r |> lineages(obscure=FALSE) |> plot()
```

Exercise 2: fit a model to a tree simulated from a different model I

```
library(pomp)
simulate("SEIRS",
  Beta=4,sigma=1,gamma=1,psi=1,omega=1,
  S0=100,E0=3,I0=5,R0=100, time=5
) -> G.seir
plot(G.seir)

G.seir |>
  sir_pomp(
    Beta=4,,gamma=1,psi=1,
    S0=100,I0=5,R0=100
  ) |> pfilter(Np=1000) |>
  replicate(n=20) |> concat() -> pf
pf |> logLik() |> logmeanexp(se=TRUE)
```

Exercise 2: fit a model to a tree simulated from a different model II

```
simulate(
  "SIR",time=10,
  Beta=3,gamma=1,psi=2,omega=1,S0=100,I0=5
) -> G.sir
plot(G.sir)

G.sir |>
  seirs_pomp(
    Beta=3,sigma=1,gamma=1,psi=2,omega=1,
    S0=100,E0=0,I0=5,R0=0
  ) |> pfilter(Np=1000) |>
  replicate(n=20) |>
  concat() -> pf

pf |> logLik() |> logmeanexp(se=TRUE)
```

References I

Asfaw, Kidus, Joonha Park, Aaron A. King, and Edward L. Ionides. 2024a. "A Tutorial on Spatiotemporal Partially Observed Markov Process Models via the R Package spatPomp." https://arxiv.org/abs/2101.01157.

Asfaw, Kidus, Joonha Park, Aaron A King, and Edward L Ionides. 2024b. "spatPomp: An R Package for Spatiotemporal Partially Observed Markov Process Models." *Journal of Open Source Software* 9 (104): 7008.

Bretó, Carles, Jesse Wheeler, Aaron A. King, and Edward L. Ionides. 2025. "panelPomp: Analysis of Panel Data via Partially Observed Markov Processes in R." https://arxiv.org/abs/2410.07934.

Ionides, Edward L., Ning Ning, and Jesse Wheeler. 2024. "An Iterated Block Particle Filter for Inference on Coupled Dynamic Systems with Shared and Unit-Specific Parameters." *Statistica Sinica* 34: 1241–62.

References II

King, Aaron A, Qianying Lin, and Edward L Ionides. 2025. "Exact Phylodynamic Likelihood via Structured Markov Genealogy Processes." *ArXiv*, arXiv–2405.

Wheeler, Jesse, AnnaElaine Rosengart, Zhuoxun Jiang, Kevin Tan, Noah Treutle, and Edward L. Ionides. 2024. "Informing Policy via Dynamic Models: Cholera in Haiti." *PLOS Computational Biology* 20 (4): 1–31. https://doi.org/10.1371/journal.pcbi.1012032.

License, acknowledgments, and links

▶ This lesson is prepared for the Simulation-based Inference for Epidemiological Dynamics module at the Summer Institute in Statistics and Modeling in Infectious Diseases, SISMID.

▶ The materials build on previous versions of this course and related courses.

▶ Licensed under the Creative Commons Attribution-NonCommercial license. Please share and remix non-commercially, mentioning its origin.

▶ Produced with R version 4.4.2 and pomp version 6.3.

▶ Compiled on 2025-07-23.

Back to Lesson
R code for this lesson