refactoring

How do we go about modifying/refactoring the code to allow it to more flexibly/generally handle different models of time-varying parameters?

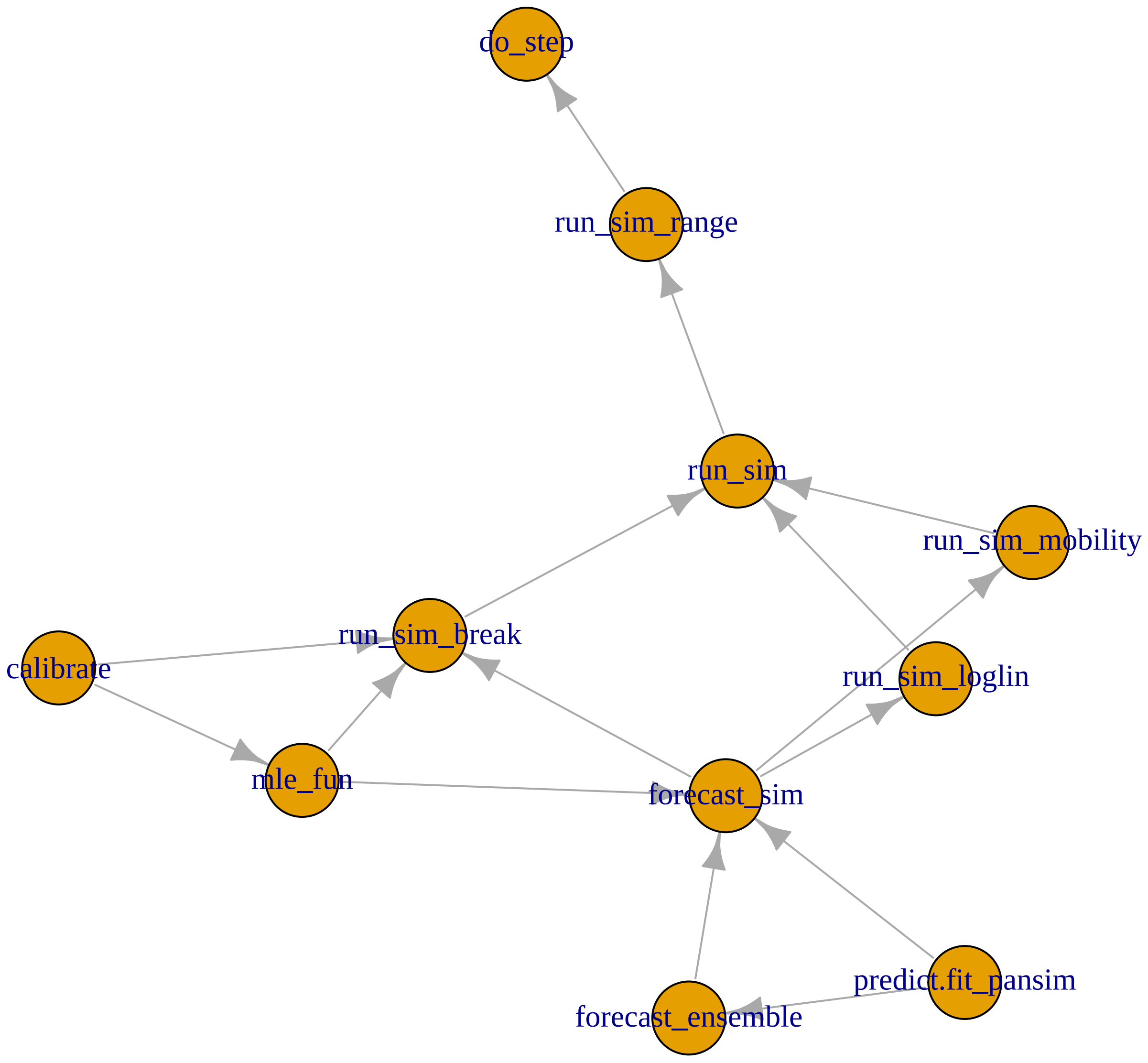
issues

- Clarity and maintainability. The more layers we have, the harder it is to figure out how everything fits together
- Performance. Things that will slow down runs (we don't care that much, except that we often run lots of sims for ensembles, estimation, etc.):
  - adding lots of compartments (e.g. Erlang-ization, age structure, spatial structure)
  - many layers of function calls, e.g. calling `run_sim_range` for every day?
- maintaining possibility to recode for speed/latent-ization (pomp, odin, Rcpp, TMB . . .)
  - avoid date manipulations in innermost loop
  - avoid character/factor processing (e.g. use `enum`-based switches if necessary)
- how do we pass information/parameters along the chain; via `...` or `do.call()` (or stored info in objects)?
- storing metadata (as attribute, or list element, or . . .)
  - attributes leave the behaviour of the original object unchanged
  - list elements are a little more transparent, but may add another layer of extraction
  - accessor/extractor methods??
  - don't even talk about S4
- post-processing of state variables
  - condensation (i.e. collapsing multiple state variables to one)
  - differencing
  - cumulating
- processing of parameters
  - unlisting/relisting
  - linking/inverse-linking

possibilities

- I'm thinking about generalizing `run_sim_break` by allowing other functions to specify the time variation of parameters (see `OTHER_TIMEFUN` below)
- collapse some of `run_sim`/`run_sim_range`/`do_step`? Calling `run_sim_range` for every step, as we would have to do with more continuously time-varying parameters, seems awkward. On the other hand, `run_sim_range` was created in the first place so that we didn't have to deal with date-processing at the lowest level. Maybe `run_sim` could process dates into a zero-based numeric format/find numeric offsets for parameters, so that `run_sim_range` was continuous but low-level (i.e. suitable for coding in TMB/Stan/etc.)?

- collapse `forecast_sim`/`run_sim_break`? This made more sense before I started thinking about adding tracks parallel to `run_sim_break` …



parameter passing

How do we deal efficiently and transparently with parameters that need to get passed/used in different places? At the moment the structure (and starting values) is stored in `opt_pars`, but different parts are used by different components. We want to save as much as necessary and pass the right pieces …

- `params` gets used only (?) within `run_sim()` and `run_sim_range()` (and in various summary methods, Jacobian calcs, etc.)
- `(log_)nb_disp` gets only used in `mle_fun`; it NULLs any elements containing `nb_disp` before passing
- `time_args` includes components

Functions

`calibrate`

takes data and a set of starting parameters/structure/etc. and uses `DEoptim` and `mle2` to estimate parameters by trajectory matching

parameters: start_date, start_date_offset, end_date, time_args, break_dates, base_params, data, opt_pars, fixed_pars, sim_fun, sim_args, aggregate_args, condense_args, priors, debug, debug_plot, debug_hist, last_debug_plot, use_DEoptim, mle2_method, mle2_control, mle2_args, seed, DE_args, DE_lwr, DE_upr, DE_cores

- `sim_args` is passed down, eventually to `run_sim`. Can we also use it for arguments to `sim_fun` (the one with time-varying stuff) and strip those arguments before passing it down - or use `...` to catch junk in `run_sim`?

`mle_fun`

- takes data and a set of starting parameters/structure/etc. and returns a negative log-likelihood (or a log-posterior-probability)
- it's useful to have this defined separately from/outside `calibrate` so that we can do other things with the log-likelihood (e.g. calculate importance weights for ensembles)

parameters: p, data, debug, debug_plot, debug_hist, opt_pars, base_params, start_date, end_date, time_args, sim_args, sim_fun, checkpoint, aggregate_args, priors, na_penalty, ...

`forecast_sim`

inverse-link transforms parameters, re-lists them into the format of `opt_pars`, then calls `run_sim_break`; then condenses/aggregates/pivots results to match data format

parameters: p, opt_pars, base_params, start_date, end_date, time_args, fixed_pars, stoch, stoch_start, sim_args, aggregate_args, condense_args, return_val, sim_fun, calc_Rt, debug, ... `p` (numeric, named parameter vector); `opt_pars` (list: starting values *and structure* for "re-listing")

`forecast_ensemble`

Calls `forecast_sim` repeatedly to generate an ensemble of trajectories, then computes (possibly weighted) quantiles (alternately may return an array of (replicate $\times$ time $\times$ variable). May also (indirectly) call `mle_fun` if importance weights are requested.

parameters: fit, nsim, forecast_args, qvec, qnames, seed, imp_wts, Sigma, scale_Sigma, calc_Rt, fix_pars_re, raw_ensembles, .progress

`run_sim_break`

Thin wrapper for `run_sim`: converts `break_dates` plus relative beta information (`rel_beta0`) into a data frame of points at which parameters change, then passes that info plus parameters to `run_sim`. `OTHER_TIMEFUN` would set up different time-dependence in parameters based on dates, parameters (and other covariates such as mobility?)

parameters: params, extra_pars, time_args, break_dates, sim_args, return_timevar, ...

`run_sim`

Constructs rate matrix; processes information about when to turn on stochasticity. Loops over change points, calling `run_sim_range` repeatedly. Currently assumes that only foi changes

parameters: params, state, start_date, end_date, params_timevar, dt, ndt, stoch, stoch_start, ratemat_args, step_args, ode_args, use_ode, condense, condense_args, verbose

`run_sim_range`

Run multiple simulation steps. Assumes constant *per capita* rates (except for foi, which is updated at each step)

parameters: params, state, nt, dt, M, ratemat_args, step_args, use_ode, ode_args

`do_step`

Run a single simulation step. Updates foi (redundantly?); generates total flows/derivatives from rate matrix and applies them

parameters: state, params, ratemat, dt, do_hazard, stoch_proc, do_exponential, testwt_scale

report variables/condensation

- how can we get all the pieces we want without re-running things?
- where is aggregation/condensation/differencing/convolving/Rt calculation getting done?
- where does obs error get added?
- do we want per-variable obs_error?

cumRep: what we want to do is have obs error on individual reporting, then cumsum() the "report" variable what we need to do is cumulate *after* applying obs error etc.

MacPan advantages/features

- formulation in terms of per capita flows (easy conversion among linear-step, hazard, ODE, etc.)
- linear subsystem
  - quick calc of current R etc.
  - simple parameter calibration to specified r, R, generation interval
  - eigenvector calculation for initialization
- calibration to any specified input stream (e.g. H, D, ICU)
- strata: cf Friston generative-model approach
  - testing
  - vaccination
  - age
- flexible time-varying parameters, with calibratable parameters
- capability for both

MacPan disadvantages

- complexity, interrelatedness
- too many options
- especially difficult/weird: when/how is condensation applied? When is expansion of states/rate matrices done?
  - possibility for automatic construction of states/rate matrices at many different levels

Refactoring goals

- core in RcppEigen/odin/TMB (speed, ability to have shrinkage/latent variables)
  - need to construct all structure as exterior, including a (sparse) rate matrix and an appropriate set of information for constructing time-varying parameters.
  - also need some kind of parameter indexing to know how the entries in the rate matrix are indexed. Sparse matrix with index into non-zero positions (similar to lme4 lambda/flexlambda structure?)

Goal

Refactor in Template Model Builder [@kristensen_tmb_2016]

Why refactor in TMB

Speed. It currently takes at least 1 hour to run a calibration of the model, sometimes longer. While this is feasible within a daily workflow, it can be problematic at crunch times (e.g., we have a PHAC request for a small change in the model to be recalibrated and returned on a short time scale); it also limits. We are in the process of adding structure (vaccination, age structure) to the model that greatly increases the size of the state space and will exacerbate our computational limitations.

If the TMB component of the model can be expanded to include the full likelihood model, we will be able to take advantage of TMB's native *automatic differentiation* capability (TMB's main design goal) to greatly improve the speed and stability of calibration.

In addition to raw speed, refactoring in TMB will allow the use of *latent variables*, i.e. vectors of parameters that are estimated with a penalization/integration step that allows them to be treated as random variables, i.e. estimated as being drawn from a distribution. This is a standard tool in statistical modeling of dynamical systems, but is only practical in Bayesian models and certain frameworks that allow estimation involving integration via the *Laplace approximation* and related approaches. One of `TMB`'s features is a built-in Laplace approximation engine, which allows any specified parameter vector to be modeled as a Gaussian latent variable. In particular it will allow us to model the transmission rate as evolving according to an autocorrelated process, which is a key element of realism. (Similar approaches are used, for example, in @asher_forecasting_2018 [which was the most accurate model in the 2018 RAPIDD Ebola forecasting challenge] and in @davies_estimated_2021 [estimating relative spread rates of COVID variants of concern].)

(**Fixme**: how close could we get to what we want with penalization, *without* implementing Laplace?)

Finally, using TMB will in principle allow an extension to estimating the model using *Hamiltonian Monte Carlo* with the `tmbstan` R package [@monnahan_no-u-turn_2018] as an interface to the Stan language [@li_fitting_2018; @chatzilena_contemporary_2019].

Why not rewrite the model entirely in C++?

- This might have some minor computational advantages, and would avoid the friction of writing a project in two different programming languages. However: (1) it is important to maintain the flexibility and accessibility of the projects to epidemiological users, who will have a much easier time modifying and customizing the functionality of the package if most of the user interface is written in R, with only the computational core implemented in TMB.

Why not use a different R/C++ interface?

The `Rcpp` interface is more widely used than `TMB`, more mature, and allows use of a broader set of C++ standard library tools. However, it does not provide easy access to the Laplace approximation or Stan HMC engine.

Basic design

(Include flow diagram/call graph here.)

Goal is to expand TMB component as far up the chain as possible, but leaving complex matching steps (rate matrix construction; construction of indices for condensing multiple state variables into single observed values; construction of data vectors for comparison with simulated dynamics; construction of date-breakpoint models or model matrices for general time variation (see below) in R, to be passed as data elements to TMB.

Basic design of MacPan works by constructing a *per capita* flow rate matrix: at each step,

- Implement `do_step/run_sim_range` in TMB: simplest (pass fixed rate matrix) will speed up all operations, but not allow leveraging latent variables or automatic differentation tools
- Implementing `run_sim`: will require construction of an *indexed* sparse matrix (i.e. a way to map changes in rate variables to changes in specific elements of the rate matrix). Again, the goal is to keep as much of the complexity in model parameterization and setup *outside* of the TMB component.
- As a next step, build a general log-linear model for parameters inside the TMB component. This will allow flexible specification of time-varying parameters (e.g. via breakpoints, splines, or smooth [logistic] transitions centered at specified breakpoints) in the calibrated model. A list of *model matrices* describing the time variation of specified parameters would be constructed in R (using the flexible R formula syntax), then passed to TMB where the model matrices could be multiplied by specified parameter vectors to determine the time variation in epidemiological parameters.