Using forcing functions in ordinary differential equations (ODEs) in Stan

Overview

Ordinary differential equations (ODEs) are widely used for mathematical modeling of natural systems. A less common but critical ODE solver input is the incorporation of fixed temporal data representing a process whose underlying equation is unknown, called a forcing function. Forcing functions are more readily supported in lower-level programming languages than in higher-level, domain-specific programming languages (DSLs) like Stan that allow fewer mathematical primitives and are more complex to extend. The sections below explain how to use a forcing function with the widely used 2-equation predator–prey ODEs example, also called Lotka–Volterra.

Preamble

First, load the libraries we will need for the entire exercise.

```
library(rstan)
library(deSolve)
library(posterior)
library(dplyr)
library(tidyr)
library(ggplot2)
```

Data science libraries of `dplyr`, `tidyr` and `ggplot2` require little introduction for many R users.

> **Note:** Those less familiar can learn about them from the freely available Carpentries lesson *R for Reproducible Scientific Analysis* (https://swcarpentry.github.io/r-novice-gapminder/) and, for a more depth, the excellent freely available book *R for Data Science* (https://r4ds.hadley.nz/).

While we our primarily goal is to use `rstan` to solve the ODEs, we use `deSolve` to generate the *ground truth* data; ground truth data here refers to generating observations with known *true* parameters so that we can assess how well the fitted parameters compare with the true parameters.

> **Note:** It is also possible to use Stan (`rstan`) itself to generate the ground truth data, as described in the Stan User's Guide > Example Models > Ordinary Differential Equations > Measurement error models. But for simplicity, we use R's popular `deSolve` package to use familiar R code where possible.

A well written Stan model should be relatively insensitive to initial conditions. The example below, demostrates a case of undesirable sensitivity to initial conditions. The `posterior` package simplifies inspecting the data wrangling work to visualize the ODE parameters accepted by Stan for each chain to troubleshoot disagreement among chains.

Typical ODEs without a forcing function

Equations in R

The Lottka–Volterra equations are:

$$\begin{aligned}\frac{dx}{dt} &= \quad \alpha x - \beta xy \\ \frac{dy}{dt} &= -\gamma y + \delta xy\end{aligned}$$

where,

State variables:

$$\begin{aligned}x &: \text{Population of prey} \\ y &: \text{Population of predators}\end{aligned}$$

Parameters:

$$\begin{aligned}\alpha &: \text{Growth rate of prey} \\ \beta &: \text{Death rate of prey due to predators} \\ \gamma &: \text{Death rate of predators} \\ \delta &: \text{Growth rate of predators due to prey}\end{aligned}$$

To code these equations in R, we need to wrap them in a function with specific function arguments common to many ODE solvers, namely, the time vector, the list of state variables, and then the list of parameters. Therefore, the input to `ode()` of the `deSolve` package is written as:

```
lv <- function(t, state, params) {
  with(as.list(c(state, params)), {
    dx <-  a * x - b * x * y
    dy <- -g * y + d * x * y
    return(list(c(dx, dy)))
  })
}
```

The `with()` function above makes the contents of the lists `state` and `params` appear in the environment without needing to explicitly refer to them with the `$` operator such as `state$x` or `params$a`; this makes the function code more legible and more closely resemble the mathematical equations. However, it's generally considered bad practice in R to use the `with()` function outside of specific functions that require unpacking many list arguments.

Generate noisy input data with known ODE parameters

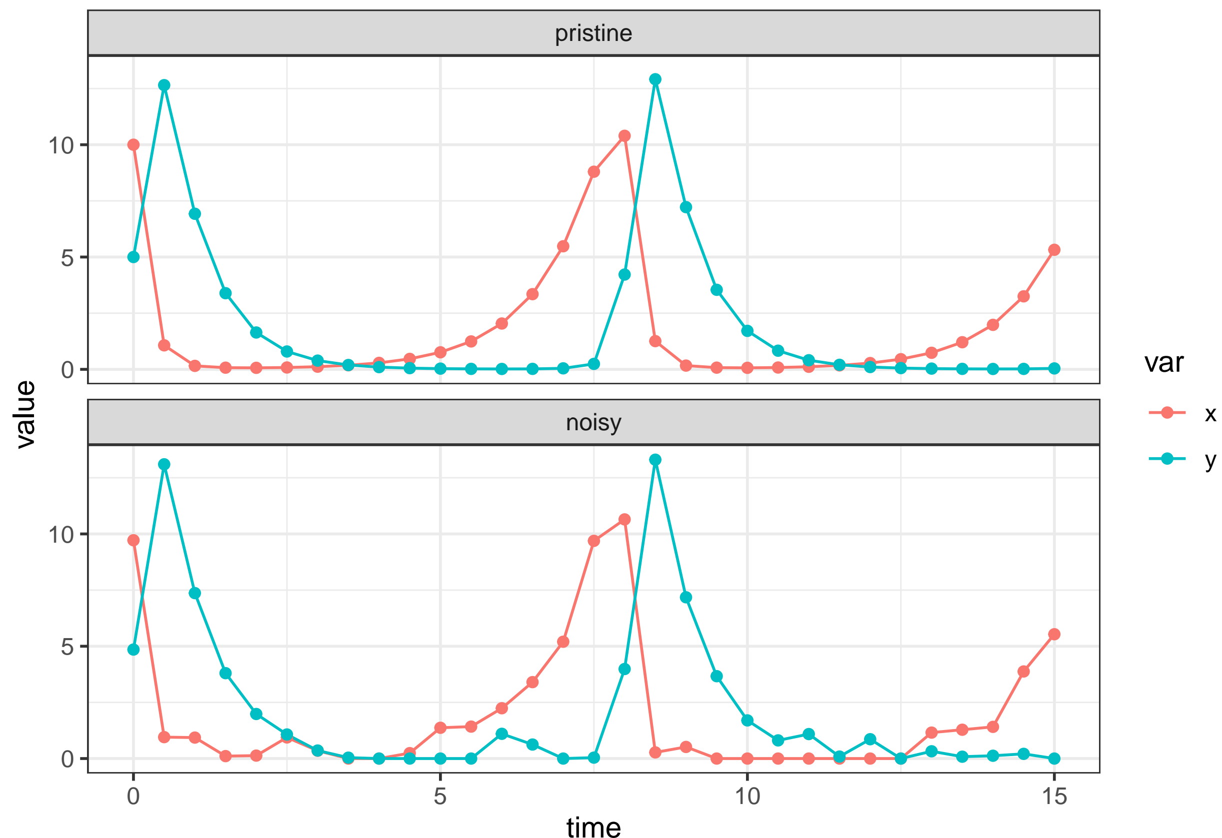
A common statistical practice is to generate data with known parameters so that we can test how well a method or model is able to recover the parameters used to generate the data. Therefore, we next simulate ODE data using known parameters `a`, `b`, `g`, and `d` (i.e. α, β, γ, and δ). Additionally, we need to assume a known initial condition x_0 and y_0 stored in the vector `true_y0`. Lastly, to make the parameter fitting more challenging and realistic to real-world observations, we add simulated noise.

```
times <- seq(0, 15, by = 0.5)
true_y0 <- c(x = 10, y = 5.0)
true_params <- c(
  a = 1.0,
  b = .5,
  g = 1.5,
  d = 0.7
)
## Solve the ODEs, then add noise to each variable using rnorm.
pristine <- ode(
  y = true_y0,
  times = times,
  func = lv,
```

```
  parms = true_params)
noisy <- pristine
set.seed(123)
noisy[, -1] <-
  pristine[, -1] + rnorm(nrow(pristine) * (ncol(pristine) - 1),
                         sd = 0.5)
noisy[noisy < 0] <- 0
```

Using `ode()` above only solves the system of equations at the timepoints specified by `times`. The simulated "pristine" and "noisy" data points are shown below, although from here on we will only use the noisy simulated data.

```
bind_rows(pristine = as.data.frame(pristine),
          noisy = as.data.frame(noisy),
          .id = "type") |>
  pivot_longer(-c(type:time), names_to = "var") |>
  mutate(type = factor(type, levels = c("pristine", "noisy"))) |>
  ggplot(aes(time, value, color = var)) +
  theme_bw() +
  facet_wrap(~ type, ncol = 1) +
  geom_line() +
  geom_point()
```



Fit the ODE parameters using the equivalent Stan model

The model we are going to fit is saved in a separate `lv.stan` file that looks like this:

```
functions{
  vector lv(real t,
        vector y,
        real a, real b, real g, real d){
    vector[2] dydt;
    //y[1] and y[2] correspond to x and y respectively.
    dydt[1] =  a*y[1] - b*y[1]*y[2];
    dydt[2] = -g*y[2] + d*y[1]*y[2];
    return dydt;
  }
}

data {
  int<lower=1> T;
  array[T] vector[2] y;
  vector<lower=0>[2] y0;
  real<lower=0> t0;
  array[T] real ts;
  real<lower=0> g;
  real<lower=0> d;
}

parameters {
  // y0 contains the initial values of x and y respectively.
  real<lower=0> a;
  real<lower=0> b;
}

model {
  array[T] vector[2] mu = ode_rk45(lv, y0, t0, ts, a, b, g, d);
  // Sample positive priors for all our ODE parameters.
  a ~ normal(0.5, 0.5);
  b ~ normal(0.5, 0.5);
  // The state variables computed by the ODE should be lognormally
  // distributed.
  for (t in 1:T) {
    y[t] ~ normal(mu[t], 1);
  }
}
```

Call the `stan()` function to supply the inputs to the model described above in the `lv.stan` file:

```
fit1 <- stan(
  model_name = "lotka-volterra",
  chains = 4,
  cores = 4,
  warmup = 1000,
  iter = 5000,
  file = "lv.stan",
  data = list(
    T = nrow(noisy) - 1,
    y = noisy[-1, c("x", "y")],
    y0 = true_y0,
    t0 = 0,
```

```
    ts = noisy[-1, "time"],
    g = true_params["g"],
    d = true_params["d"]),
  seed = 12345
)
```

```
## Warning: The largest R-hat is 2.41, indicating chains have not mixed.
## Running the chains for more iterations may help. See
## https://mc-stan.org/misc/warnings.html#r-hat
```

```
## Warning: Bulk Effective Samples Size (ESS) is too low, indicating posterior means and medians may be
## Running the chains for more iterations may help. See
## https://mc-stan.org/misc/warnings.html#bulk-ess
```

```
## Warning: Tail Effective Samples Size (ESS) is too low, indicating posterior variances and tail quant
## Running the chains for more iterations may help. See
## https://mc-stan.org/misc/warnings.html#tail-ess
```

We get warnings about how our initial conditions were not able to agree and mix together that we will explore in the next section.

Inspect the Stan fit

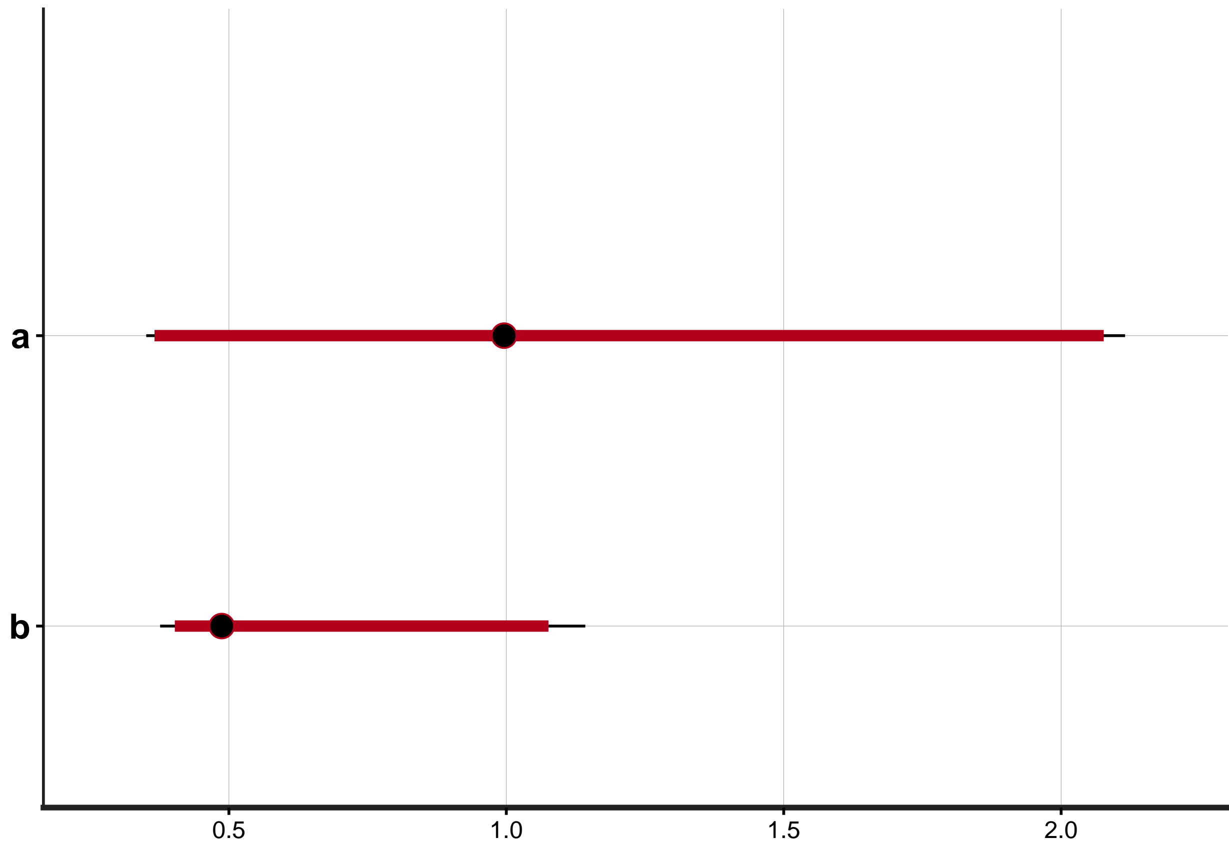
```
## Numerical summary of the fits.
fit1
```

```
## Inference for Stan model: anon_model.
## 4 chains, each with iter=5000; warmup=1000; thin=1;
## post-warmup draws per chain=4000, total post-warmup draws=16000.
##
##         mean se_mean     sd    2.5%     25%     50%   75% 97.5% n_eff   Rhat
## a       1.11    0.43   0.61    0.35    0.83    1.00  1.27  2.12     2  30.48
## b       0.61    0.19   0.27    0.38    0.45    0.49  0.66  1.14     2   7.04
## lp__ -198.85  150.29 212.55 -520.56 -333.77 -137.53 -6.28 -5.62     2 210.97
##
## Samples were drawn using NUTS(diag_e) at Mon Jun  1 17:51:46 2026.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

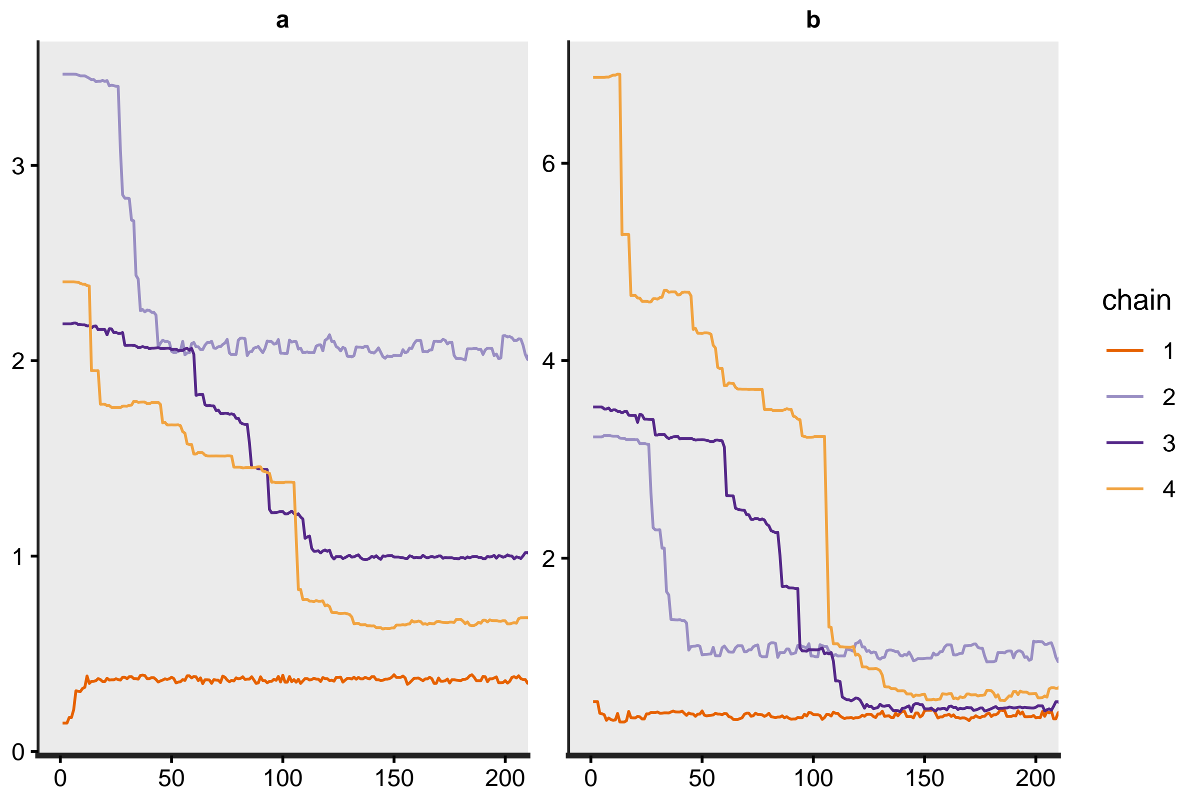
```
## Parameter plot of the fits.
plot(fit1)
```

```
## ci_level: 0.8 (80% intervals)
```

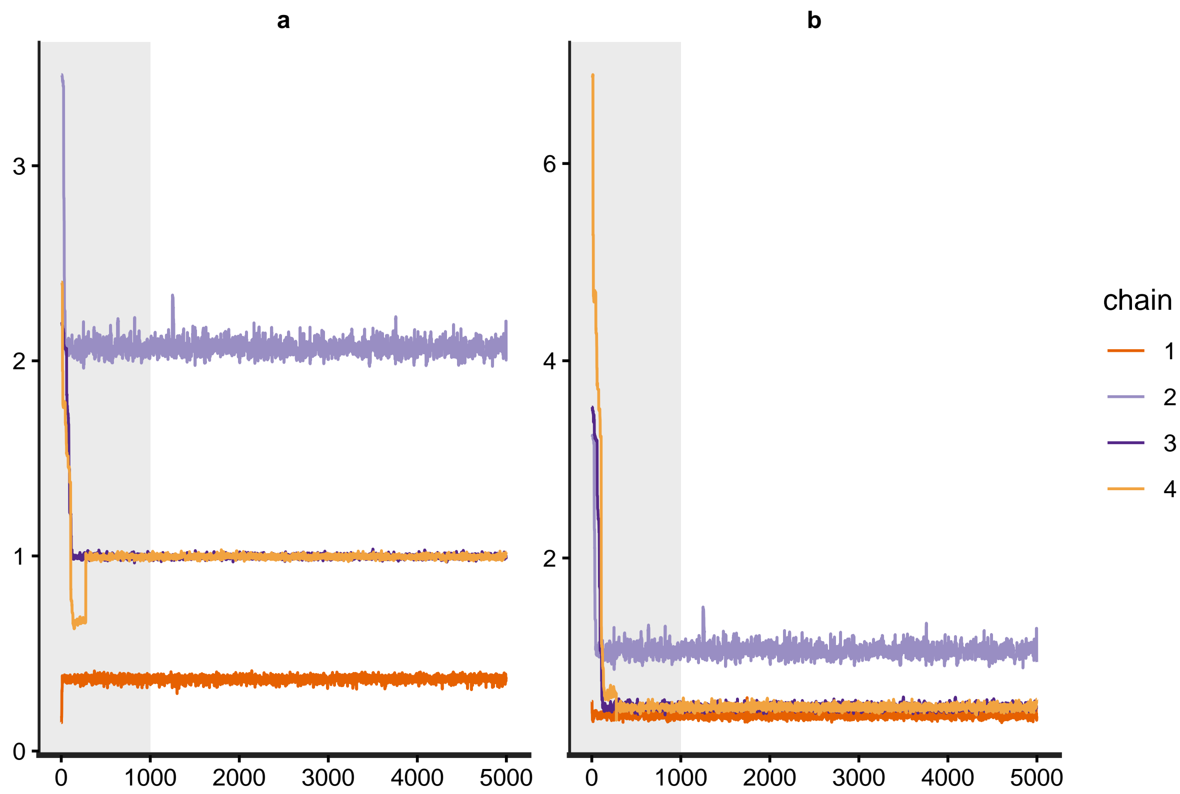
```
## outer_level: 0.95 (95% intervals)
```




```
## Find initial values creating problems with chains converging.
traceplot(fit1, inc_warmup = TRUE, window = c(0, 200))
```

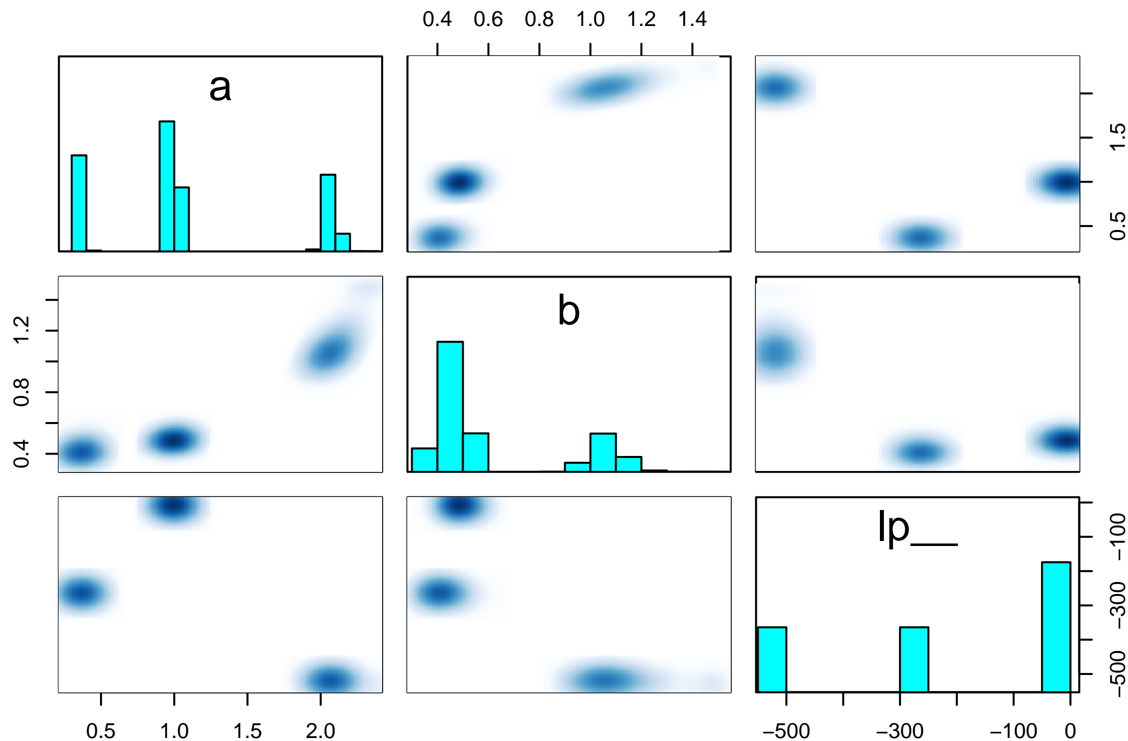


```r
## Show all samples including warmup.
traceplot(fit1, inc_warmup = TRUE)
```




```r
## Show the pairs plot.
pairs(fit1, pars = c("a", "b", "lp__"))
```

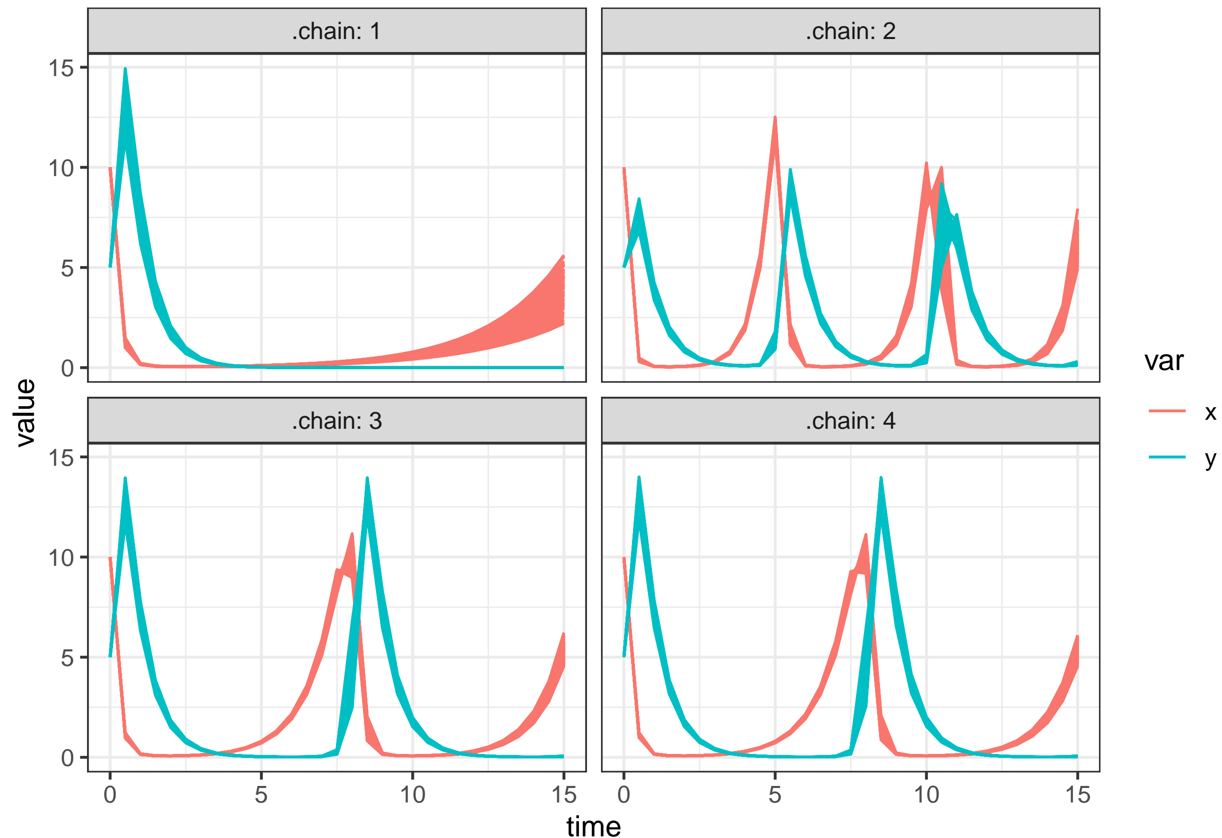
```
## Warning in par(usr): argument 1 does not name a graphical parameter
## Warning in par(usr): argument 1 does not name a graphical parameter
## Warning in par(usr): argument 1 does not name a graphical parameter
```




```
## Plot ODEs using a sample of 100 parameters from each of the chains.
n <- 100
set.seed(123)
posterior::as_draws_df(fit1) |>
  select(.chain, .iteration, a, b) |>
  group_by(.chain) |>
  ## Sample greater variation by only choosing one sample from each ntile.
  mutate(qa = ntile(a, n / 5),
         qb = ntile(b, n / 5)) |>
  group_by(.chain, qa, qb) |>
  sample_n(1) |>
  ## Now sample the 100 parameters from each chain.
  group_by(.chain) |>
  sample_n(n) |>
  ungroup() |>
  rowwise() |>
  mutate(mat = list(
    ode(
      y = true_y0,
      times = times,
      func = lv,
      parms = list(
        a = a,
        b = b,
        g = true_params["g"],
        d = true_params["d"])) |>
      as.data.frame())) |>
  unnest(cols = "mat") |>
  ungroup() |>
  pivot_longer(-c(.chain:time), names_to = "var") |>
```

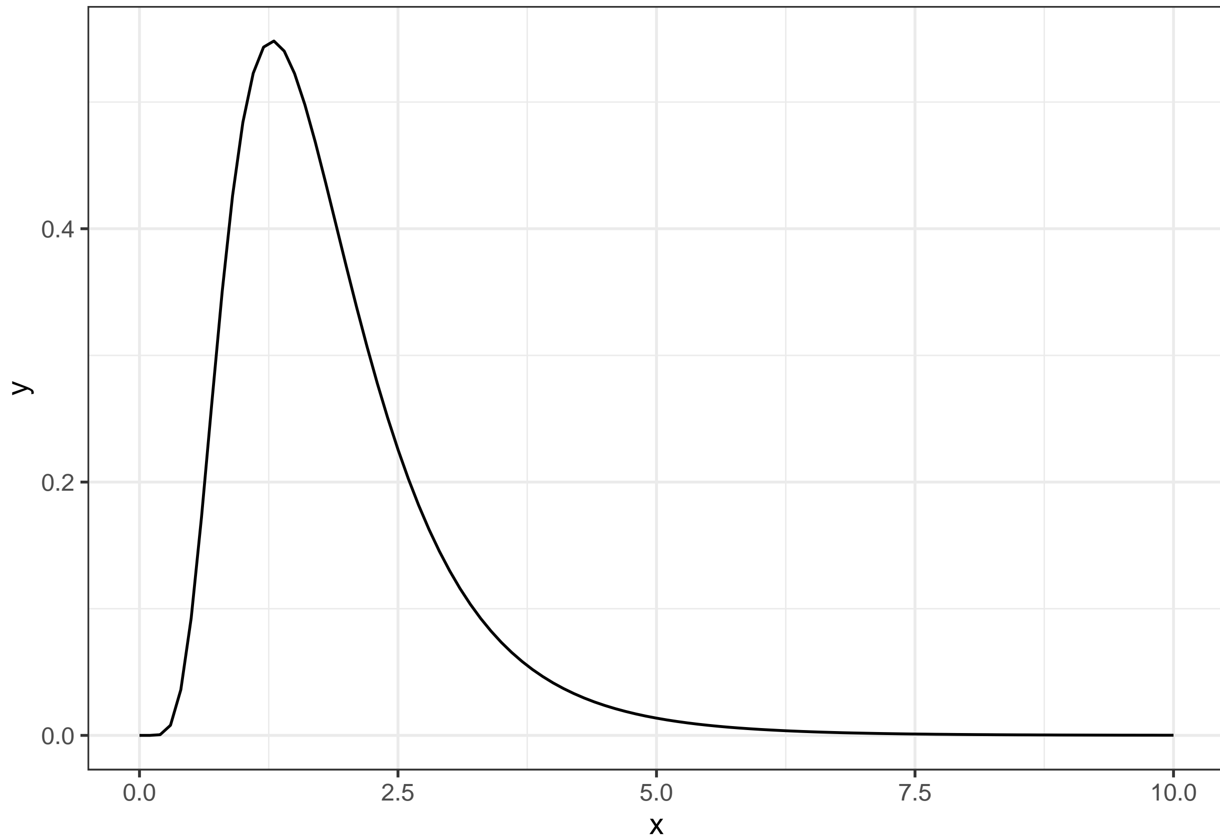
```r
  mutate(.chain = as.factor(.chain)) |>
  ggplot(aes(x = time,
             y = value,
             group = interaction(.chain, .iteration, var),
             color = var)) +
  theme_bw() +
  facet_wrap(~ .chain, labeller = "label_both") +
  geom_line()
```

```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```



In the last plot above that shows 100 ODE samples simulated by each of the 4 chains, we see that one of the chains is not able to capture the true periodicity. This is because of two reasons: (1) the oscillatory ODE system contains "peaks" of higher posterior probabilities and moving between these "peaks" is hard and (2) we are using large priors `a ~ normal(0.5, 0.5)` and `b ~ normal(0.5, 0.5)`:

```r
## Plot the priors for `a` and `b` used in the Stan model.
data.frame(x = seq(0, 10, .1)) |>
  ggplot(aes(x)) +
  theme_bw() +
  stat_function(fun = dlnorm,
                args = c(meanlog = 0.5,
                         sdlog = 0.5))
```



But the bottom row of the pairs plot further above shows that the values of `b` (β) around 0.5 have the highest `lp__` (log-probability) and that values of `a` (α) around 1.0 have the highest `lp__`; 0.5 and 1.0 are indeed their true values. We often learn more when things go "wrong" and hopefully reading these diagnostic plots has helped you learn a little more about troubleshooting models that do not perfectly fit the noisy data!

ODE with a forcing function

Modified equations in R

Finally! We get to discuss the main topic about how to use a forcing function when the equation of a state variable is unknown. Let's say that we did not know the equation terms for the predator state variable, `y`. Then, instead of two ODEs, we need to solve a single ODE. But we must replace the `y` term in the `x` equation with a fixed trajectory of `y`.

Before we had:

$$\begin{aligned}\frac{dx}{dt} &= \quad \alpha x - \beta x y \\ \frac{dy}{dt} &= -\gamma y + \delta x y\end{aligned}$$

Now we have only the first equation and pretend we do not yet know the process that governs the second equation:

$$\frac{dx}{dt} = \quad \alpha x - \beta x y$$

We cannot fix `y` as a single value; `y` must vary in some way because `x` varies and `y` and `x` influence eachother. So we must supply a fixed trajectory of `y` based on the noisy data that only depends on time because `y` is no longer part of the system of equations that the ODE solver has access to. We must define an external function that produces a fixed trajectory of `y`.

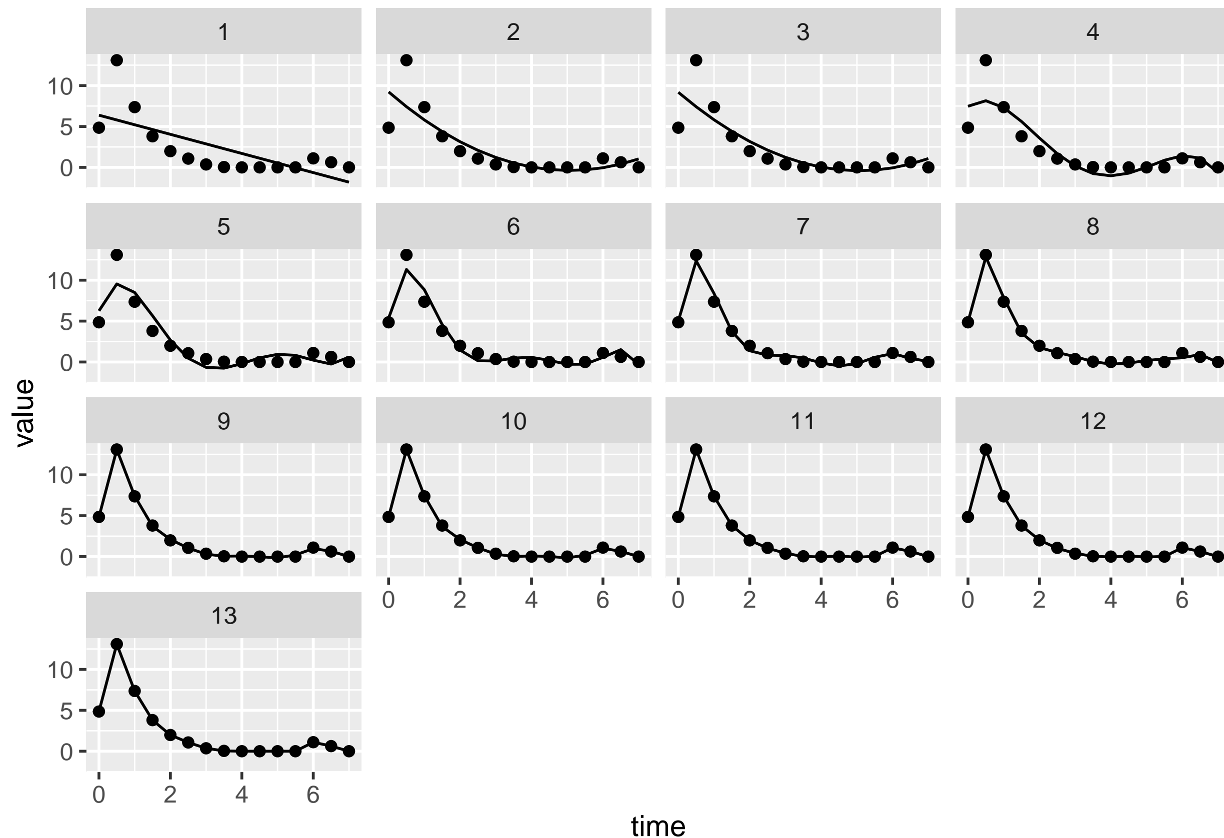
Ideally, we would use a spline function. Splines are wonderful mathematical funtions because they're differentiable while still allowing us to fit complex, squiggly looking curves. However, spline functions are numerically harder to implement and Stan does not yet provide the mathematical primitives to easily add them. Therefore, we will use a simpler polynomial function:

$$
\begin{aligned}
\frac{dy}{dt} &= c_n t^n + c_{n-1} t^{n-1} \ldots + c_2 t^2 + c_1 t^1 + c_0 t^0 \\
\implies \frac{dy}{dt} &= \sum_{i=0}^{n} c_i t^i
\end{aligned}
$$

Where, the polynomial terms are $t_0, t_1, t_2, \ldots, t_{n-1}, t_n$ and their corresponding coefficients are $c_0, c_1, c_2 \ldots, c_{n-1}, c_n$.

> **Note:** Unlike a spline function, a polynomial function has trouble fitting periodic data. See the plot below.

```r
## Plot different degrees of polynomials fit to the prey data.
max_time <- 7
max_degree <- length(times[times <= max_time]) - 2
fit_poly <- list()
for (i in seq_len(max_degree)) {
  fit_poly[[i]] <-
    lm(y ~ poly(time, i),
       data =
         noisy[, c("time", "y")] |>
         as.data.frame() |>
         filter(time <= max_time))
}
noisy |>
  as.data.frame() |>
  filter(time <= max_time) |>
  mutate(across(time,
                setNames(lapply(seq_len(max_degree),
                                function(i) {
                                  return(function(time) predict(fit_poly[[i]]))
                                }),
                         paste0("y_poly_", seq_len(max_degree))))) |>
  pivot_longer(-c(time:y), names_to = "degree") |>
  mutate(degree =
           sub("time_y_poly_", "", degree) |>
           as.integer()) |>
  ggplot(aes(time, value)) +
  facet_wrap(~ degree) +
  geom_line() +
  geom_point(aes(y = y))
```

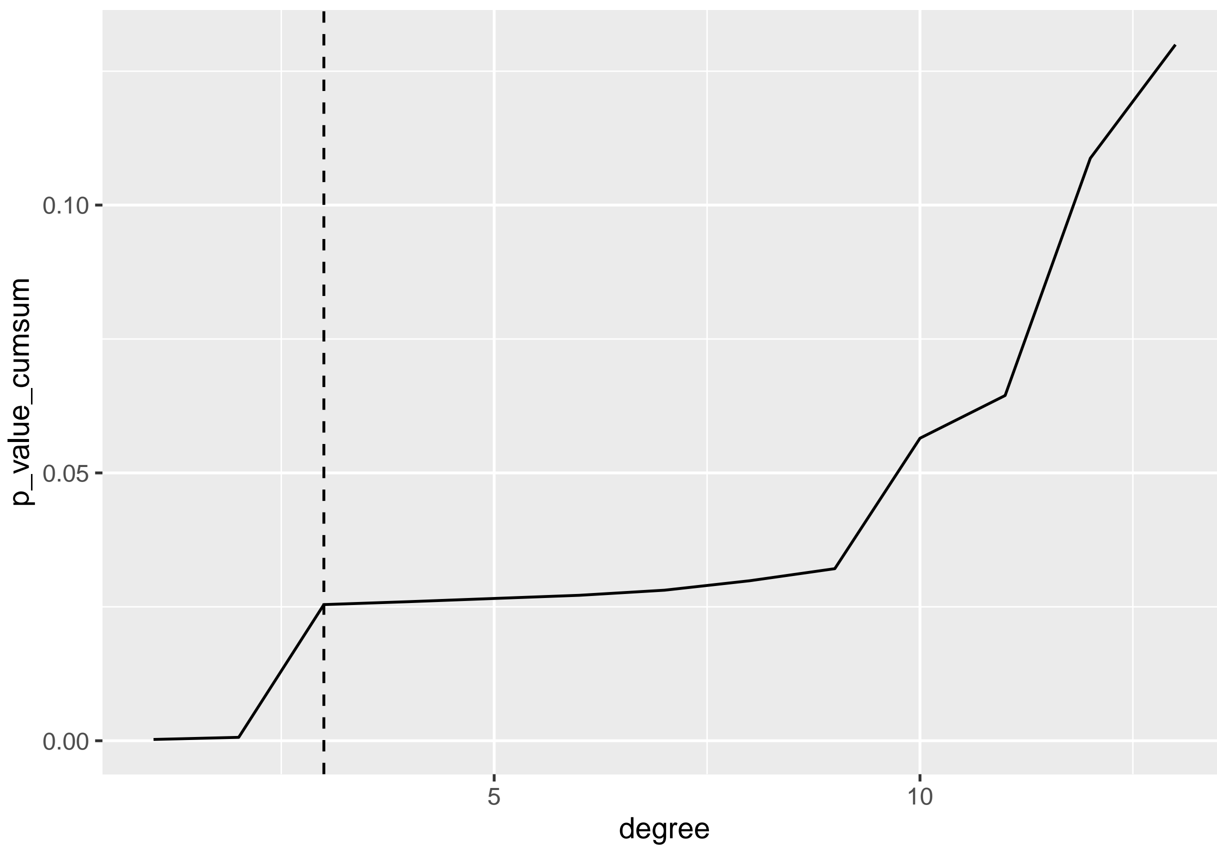



```r
## T-test showing other than the intercept, terms up to and including degree
## that captures the most variance.
summary(fit_poly[[length(fit_poly)]])
```

```
##
## Call:
## lm(formula = y ~ poly(time, i), data = filter(as.data.frame(noisy[,
##     c("time", "y")]), time <= max_time))
##
## Residuals:
##          1          2          3          4          5          6          7
## -5.901e-07  8.262e-06 -5.370e-05  2.148e-04 -5.907e-04  1.181e-03 -1.772e-03
##          8          9         10         11         12         13         14
##  2.025e-03 -1.772e-03  1.181e-03 -5.907e-04  2.148e-04 -5.370e-05  8.262e-06
##         15
## -5.901e-07
##
## Coefficients:
##                   Estimate Std. Error  t value Pr(>|t|)
## (Intercept)      2.2865369  0.0009651  2369.31 0.000269 ***
## poly(time, i)1  -9.7887841  0.0037377 -2618.95 0.000243 ***
## poly(time, i)2   6.0191981  0.0037377  1610.41 0.000395 ***
## poly(time, i)3   0.0959758  0.0037377    25.68 0.024780 *
## poly(time, i)4  -4.3403389  0.0037377 -1161.24 0.000548 ***
## poly(time, i)5   3.9692103  0.0037377  1061.94 0.000599 ***
## poly(time, i)6  -4.0371723  0.0037377 -1080.13 0.000589 ***
## poly(time, i)7   2.4978560  0.0037377   668.29 0.000953 ***
```

```
## poly(time, i)8  -1.3501600  0.0037377  -361.23 0.001762 **
## poly(time, i)9   1.0565859  0.0037377   282.69 0.002252 **
## poly(time, i)10  0.0976533  0.0037377    26.13 0.024355 *
## poly(time, i)11  0.2982591  0.0037377    79.80 0.007977 **
## poly(time, i)12  0.0536424  0.0037377    14.35 0.044287 *
## poly(time, i)13  0.1121392  0.0037377    30.00 0.021211 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.003738 on 1 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:      1
## F-statistic: 1.059e+06 on 13 and 1 DF,  p-value: 0.0007608
```

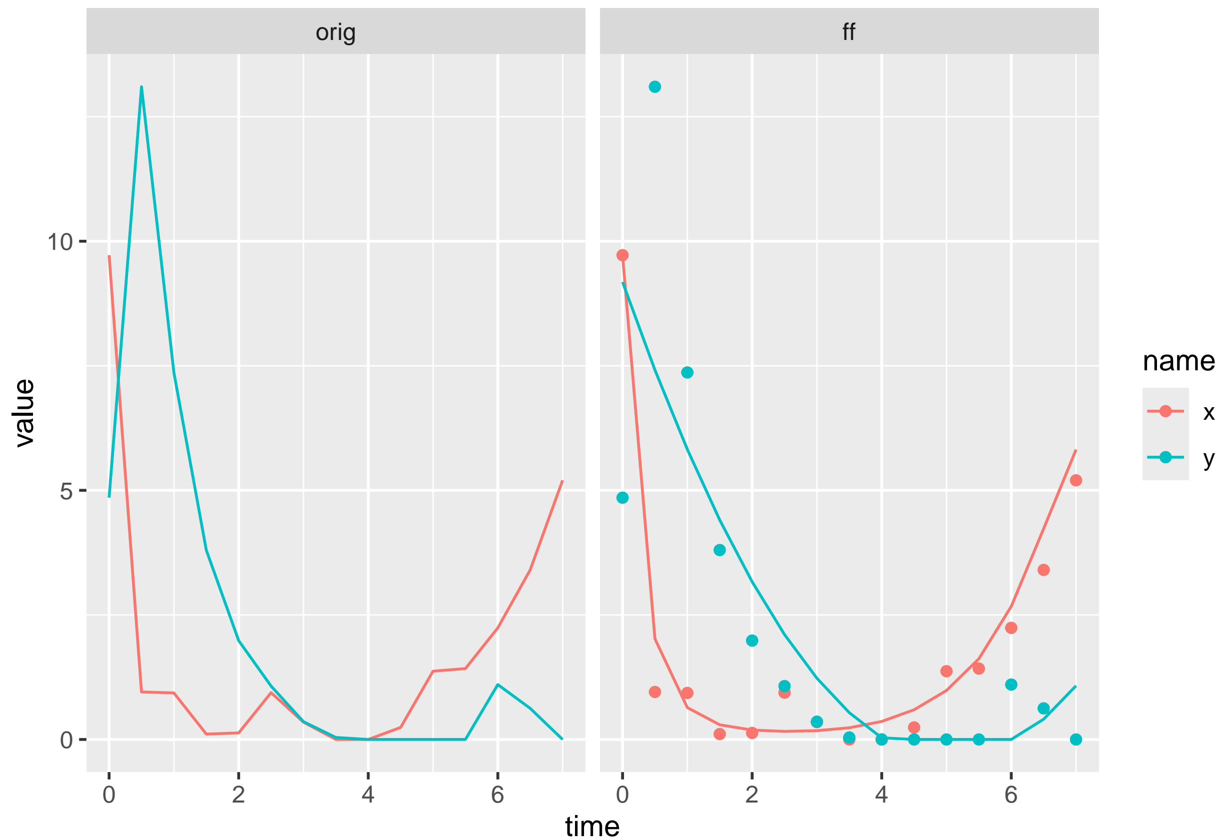
```
## Choose our polynomial degree based on the first p.value elbow.
degrees <-
  summary(fit_poly[[length(fit_poly)]]) |>
  broom::tidy() |>
  filter(term != "(Intercept)") |>
  mutate(p_value_cumsum = cumsum(p.value),
         degree = gsub(term, pattern = ".*[)]", replacement = "") |>
           as.integer())
degree <- 3
degrees |>
  ggplot(aes(x = degree, y = p_value_cumsum)) +
  geom_line() +
  geom_vline(xintercept = degree, linetype = 2)
```



```r
## Workaround bug in non-standard evaluation of predict.lm().
i <- degree
```

```r
## Now generate the ODE data using the forcing function (using t) and
## ignore any input for predator (dy).
y <- function(t) {
  predator <- predict(fit_poly[[degree]], data.frame(time = t))
  if (predator < 0) {
    return(0)
  }
  predator
}
lv_poly <- function(t, state, params) {
  with(as.list(c(state, params)), {
    ## Save computation time by not recalculating y(t) everytime.
    y_ <- y(t)
    dx <- a * x - b * x * y_
    return(list(c(dx)))
  })
}
```

```r
fit_ode_ff <- ode(
  y = noisy[1, "x"],
  times = times[times <= max_time],
  func = lv_poly,
  parms = true_params[c("a", "b")])
noisy |>
  as.data.frame() |>
  filter(times <= max_time) |>
  rename_with(~ paste0(.x, "_orig"), -time) |>
  bind_cols(x_ff = fit_ode_ff[, 2]) |>
  bind_cols(y_ff = sapply(times[times <= max_time], y)) |>
  pivot_longer(-time) |>
  separate_wider_delim(name, delim = "_", names = c("name", "type")) |>
  mutate(type = factor(type, levels = c("orig", "ff"))) |>
  ggplot(aes(time, value, group = name, color = name)) +
  facet_wrap(~ type) +
  geom_line() +
  geom_point(data = . %>%
               filter(type == "orig") %>%
               mutate(type = factor("ff", levels = c("orig", "ff"))))
```



Fit the ODE parameters using the equivalent modified Stan model

We implement these equations in the Stan code.

The model we are going to fit is saved in a separate `lv-forcing-function.stan` file that looks like this:

```
functions {
  /* Forward declaration of prey forcing function. */
  real Y(real t, int N_coef, vector coef);

  /* Lotka-Volterra ODE equations with predator forcing function.

     See https://en.wikipedia.org/wiki/Lotka%E2%80%93Volterra_equations

     @param t The single time point at which to evaluate the ODE.
     @param x State variables of populations.
     @param a Prey growth rate parameter (alpha).
     @param b Prey death rate parameter from the presence of predators (beta).
     @return Left-hand side (LHS) of the system of ODE equations (prey only).
   */
  vector lv(real t,
         vector x,
         real a,
         real b,
         int N_coef,
         vector coef) {
    vector[1] dx;
    real y_ = Y(t, N_coef, coef);
```

```
    dx[1] = a * x[1] - b * x[1] * y_;
    return dx;
  }

  /* Predator forcing function.

     The original predator data is approximated as a continuous,
     differentiable function here using a polynomial because it's
     simple to implement in Stan; ideally one would want to use a more
     precise spline, but that's more difficult.  Implementing a spline
     can be done later once the model is working.

     @param t Single time point at which to evaluate the forcing
              function.
     @return Predator value at time t.
   */
  real Y(real t, int N_coef, vector coef) {
    real predator = 0;
    for (i in 1:N_coef) {
      predator += coef[i] * pow(t, i-1);
    }
    /* Set negative values to zero, otherwise the calculation will fail.*/
    if (predator < 0) {
      return 0;
    }
    return predator;
  }
}

data {
  /* Dimensions. */
  int<lower=1> T;
  int<lower=1> V;

  /* Experimental data. */
  array[T] real ts;
  array[T] vector[V] y;

  /* Coefficients of fitted polynomial. */
  int<lower=1> N_coef;
  vector[N_coef] coef;

  /* Initial conditions. */
  real<lower=0> t0;
  vector<lower=0>[V] y0;
}

parameters {
  /* Prey death rate. */
  real<lower=0> b;
  /* Prey growth rate; must be higher than prey death rate. */
  real<lower=b> a;
}
```

```
model {
  array[T] vector[V] mu = ode_rk45(lv, y0, t0, ts, a, b, N_coef, coef);
  // Sample positive priors for all our ODE parameters; the T[0, ]
  // truncates the normal distrubiton at 0 so that negative values are
  // not sampled.
  a ~ normal(0.5, 0.5);
  b ~ normal(0.5, 0.5);
  // The state variables computed by the ODE should be normally
  // distributed.
  for (t in 2:T) {
    y[t] ~ normal(mu[t], 1);
  }
}
```

```
## Run Stan using the same forcing function.
fit_stan <-
  stan(
    model_name = "lotka-volterrs-forcing-function",
    chains = 4,
    cores = 4,
    warmup = 1000,
    iter = 5000,
    file = "lv-forcing-function.stan",
    data = list(
      ## Must remove the row containing timepoint 0 and instead
      ## provide it with the initial conditions.
      ##
      ## Dimensions:
      T = sum(times > 0 & times <= max_time),
      V = ncol(noisy) - 2L,
      ## Experimental data:
      ts = times[times > 0 & times <= max_time],
      y = noisy[which(times > 0 & times <= max_time), "x", drop = FALSE],
      ## Coefficients of fitted polynomial:
      N_coef = length(coef(fit_poly[[degree]])),
      coef = coef(fit_poly[[degree]]),
      ## Initial conditions:
      t0 = 0,
      y0 = as.array(noisy[1, "x"])),
    seed = 123
)
```

Inspect the Stan forcing function fit

```
fit_stan
```

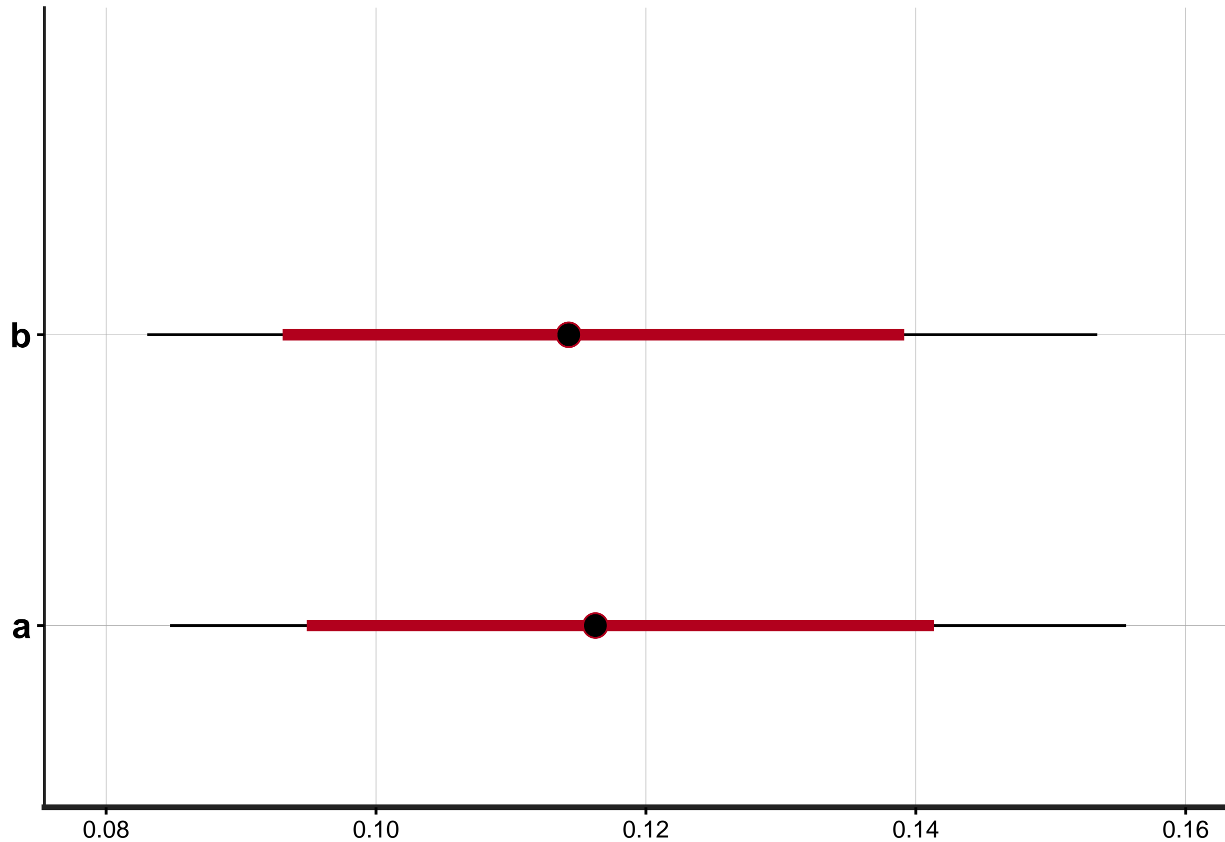
```
## Inference for Stan model: anon_model.
## 4 chains, each with iter=5000; warmup=1000; thin=1;
## post-warmup draws per chain=4000, total post-warmup draws=16000.
##
##          mean se_mean   sd    2.5%     25%     50%     75%   97.5% n_eff Rhat
## b        0.12    0.00 0.02    0.08    0.10    0.11    0.13    0.15  7910    1
## a        0.12    0.00 0.02    0.08    0.10    0.12    0.13    0.16  7942    1
```

```
## lp__ -188.70    0.01 1.06 -191.51 -189.12 -188.38 -187.93 -187.64  5816    1
##
## Samples were drawn using NUTS(diag_e) at Mon Jun  1 17:52:04 2026.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

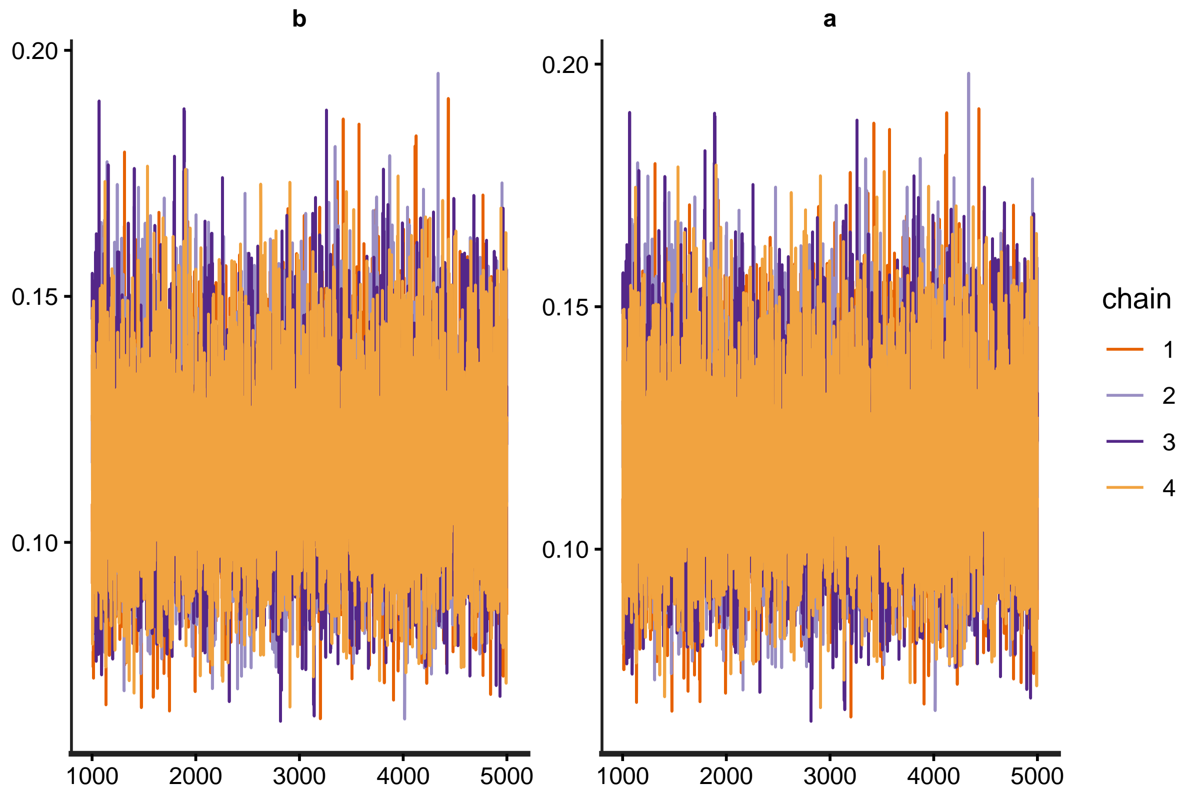
```
plot(fit_stan)
```

```
## ci_level: 0.8 (80% intervals)
```

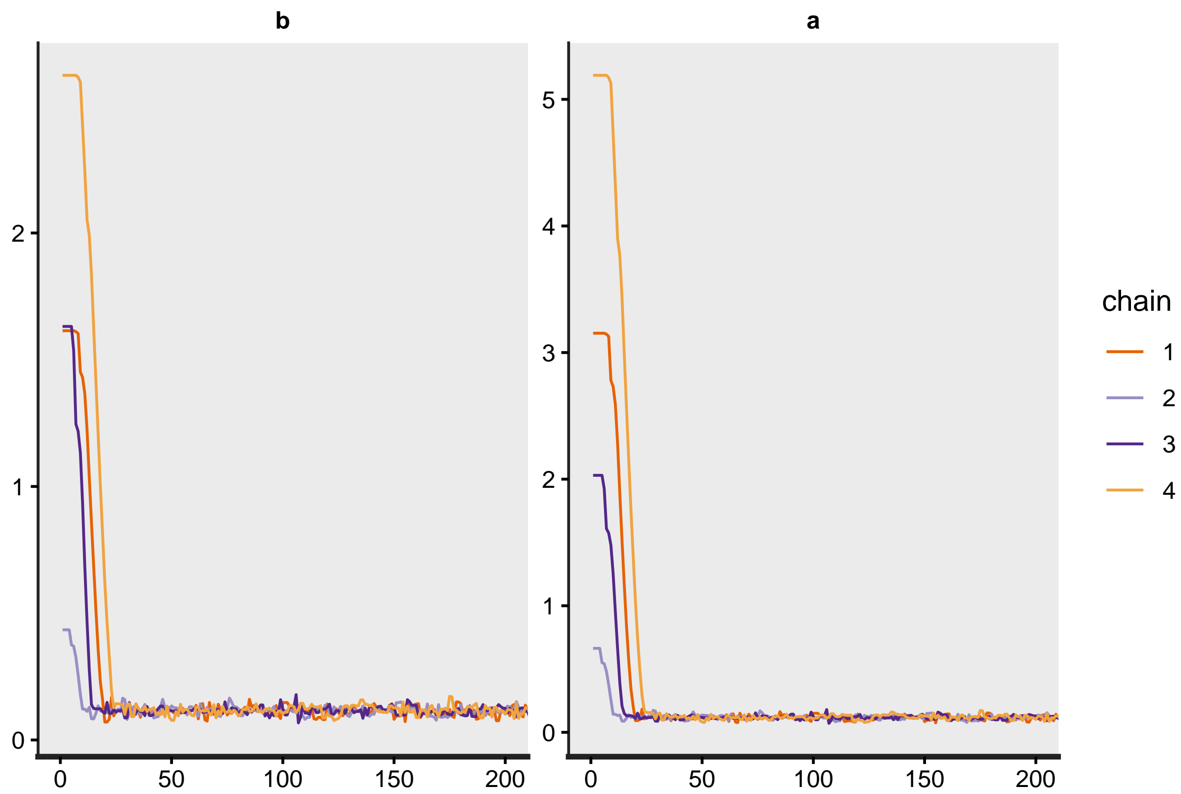
```
## outer_level: 0.95 (95% intervals)
```




```
traceplot(fit_stan)
```

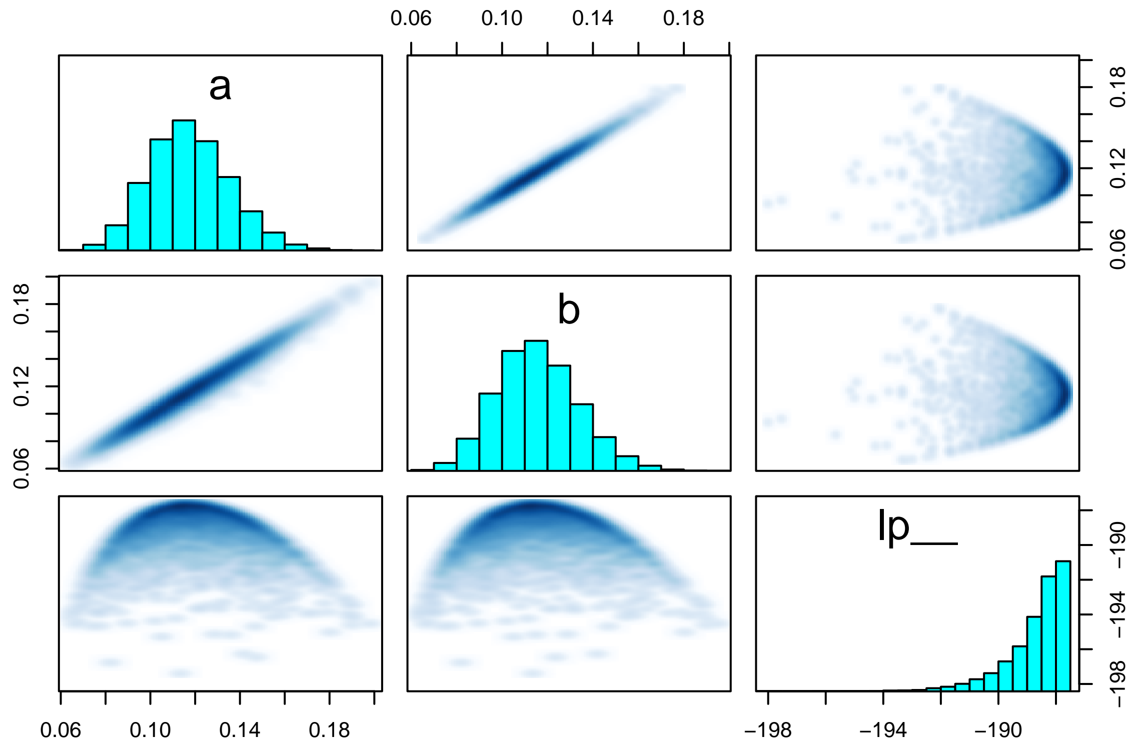



```
traceplot(fit_stan, inc_warmup = TRUE, window = c(0, 200))
```



```r
pairs(fit_stan, pars = c("a", "b", "lp__"))
```

```
## Warning in par(usr): argument 1 does not name a graphical parameter
## Warning in par(usr): argument 1 does not name a graphical parameter
## Warning in par(usr): argument 1 does not name a graphical parameter
```




```r
## Plot ODEs using a sample of 100 parameters from each of the chains.
set.seed(123) # For sample_n below.
## Running the ODEs here takes about minute to run (4 chains x 100 samples x
## 70 timepoints).
n <- 100
system.time({
  df_stan_fit <-
    posterior::as_draws_df(fit_stan) |>
    select(.chain, .iteration, a, b) |>
    group_by(.chain) |>
    ## Sample greater variation by only choosing one sample from each ntile.
    mutate(qa = ntile(a, n / 4),
           qb = ntile(b, n / 4)) |>
    group_by(.chain, qa, qb) |>
    sample_n(1) |>
    ## Now sample the 100 parameters from each chain.
    group_by(.chain) |>
    sample_n(n) |>
    ungroup() |>
    rowwise() |>
    mutate(mat = list(
      ode(
        y = true_y0[1],
        times = times[times <= max_time],
```

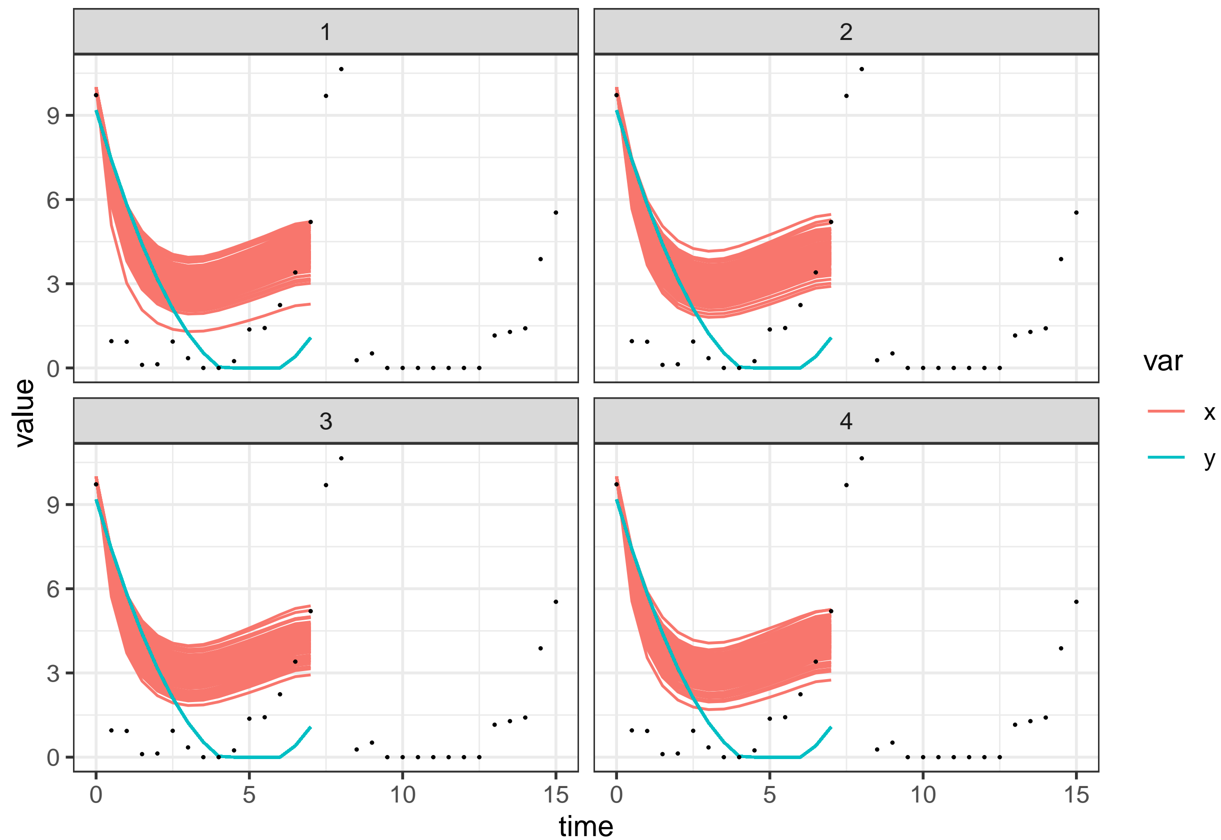
```r
        func = lv_poly,
        parms = list(
          a = a,
          b = b)) |>
        as.data.frame())) |>
    unnest(cols = "mat") |>
    rowwise() |>
    mutate(y = y(time)) |>
    ungroup() |>
    pivot_longer(-c(.chain:time), names_to = "var") |>
    mutate(.chain = as.factor(.chain))
})
```

```
## Warning: Dropping 'draws_df' class as required metadata was removed.
```

```
##    user  system elapsed
##  31.239   0.001  31.249
```

```r
## Plot the fitted ODEs from Stan.
df_stan_fit |>
  ggplot(aes(x = time,
             y = value,
             group = interaction(.chain, .iteration, var),
             color = var)) +
  theme_bw() +
  facet_wrap(~ .chain) +
  geom_line() +
  ## Show the predator data provided to Stan.
  geom_point(
    data =
      noisy[, c("time", "x")] |>
      as.data.frame() |>
      pivot_longer(-time),
    group = NULL,
    color = NULL,
    size = 0.1)
```

```
## Warning in geom_point(data = pivot_longer(as.data.frame(noisy[, c("time", :
## Ignoring empty aesthetics: `group` and `colour`.
```



Finishing thoughts

1. Don't simply choose a forcing function only based on how it fits its own data; instead, choose a forcing function by how well the other equations fit their data by running the full ODE system.
2. Spline functions are great as forcing functions, but not always practical to use. Polynomials can work, but may not work well with some types of periodic data; subset the data to timepoints where the fit looks good.
3. You may have noticed `set.seed(...)` used in a few places. This is to make the stochastic code outputs reproducible, but one would typically not use these; it is more common to use store the value of `.Random.seed` to record a seed used for a simulation than to force a specific outcome because well-adjusted simulations should produce very similar results in spite of pseudo-random number generator initial states.

References