Examples: Nonlinear continuous-time models

Lu Ou, Michael D. Hunter, Sy-Miin Chow

September 16, 2018

This file demonstrates the utilization of **dynr** in fitting single-regime and regime-switching nonlinear dynamic models. As extensions of linear models, nonlinear dynamic models incorporate nonlinearities into the change processes. Such nonlinearities may take the form of interactions between components of a system, and have many useful applications across different scientific disciplines. In the study of human dynamics, for instance, many processes are characterized by changes that are dependent on interactions with other processes. Nonlinear ordinary differential equations have been used to model, among other phenomena, ovulatory regulation [**Boker2014**], circadian rhythms [**Brown99b**], cerebral development [**Thatcher98a**], substance use [**Boker1998**], cognitive aging [**Chow04a**], parent-child interactions [**Thomas76a**], couple dynamics [**Chow07a**, **Gottman2002**]; and sudden transitions in attitudes [**vanderMaas03a**]. To facilitate the specification of more complex dynamic models, especially those that involve the use of specialized mathematical functions (e.g., trigonometric, power, logistic and exponential functions), or those for which the user would rather not specify in matrix form, **dynr** provides users with a formula interface that can accommodate nonlinear as well as linear dynamic functions.

1 Single-regime nonlinear continuous-time model

To illustrate the use of the formula interface in **dynr**, we use a benchmark nonlinear ordinary differential equation model, the predator-prey model [**Lotka25a**, **Volterra26a**, **Hofbauer88a**]. The predator-prey model is a classic model for representing the nonlinear dynamics of interacting populations or components of any system of interest. In this model, there are two populations, one of predators (e.g., foxes) and another of prey (e.g., rabbits). The food supply of the prey is assumed to be unbounded, but the food supply of the predators is the prey. As the predator population grows, they decrease the prey population. Consequently, as the prey population shrinks, the predator population must also decrease with its diminishing food supply. The most often cited behavior of the predator-prey system while in a particular parameter range is ongoing oscillations in the predator and prey populations with a phase lag between them.

The utility of the predator-prey model extends far beyond the area of population dynamics. Direct applications or extensions of this predator-prey system

include the epidemic models of the onset of social activities (EMOSA) used to study the spread of smoking, drinking, delinquency, and sexual behaviors among adolescents [**EMOSA93**, **EMOSA98**], the cognitive aging model [**Chow04a**], and the model of couples' affect dynamics [**Chow07a**]. In the EMOSA, smokers (predators) may interact with non-smokers (prey) to produce varying numbers of smokers and nonsmokers over time depending on the parameters of the system. Likewise, romantic couples may mutually drive their partners' affective states through ongoing interactions with each other, creating novel and testable hypotheses about human behavior.

Written as a differential equation, the predator-prey model is expressed as:

$$d(prey(t)) = (a\ prey(t) - b\ prey(t)\ predator(t))\, dt \tag{1}$$

$$d(predator(t)) = (-c\ predator(t) + d\ prey(t)\ predator(t))\, dt \tag{2}$$

where the parameters a, b, c, d are all constrained to be greater than or equal to 0. These equations make up the continuous-time dynamics for this system (i.e., the special case of Equation 3 for this model). Examining the prey equation (Equation 1), the prey population would increase exponentially without bound if there were zero predators. Similarly, examining the predator equation (Equation 2), if the prey population was zero, then the predator population would decrease exponentially to zero (i.e., go extinct).

For demonstration purposes, we have included with the **dynr** package a set of simulated data generated with true parameter values: $a = 2, b = 1, c = 4, d = 1, e = .25, f = 5$. A fully functional demo script can be found as one of the demos in **dynr** using:

```
> file.edit(system.file("demo", "NonlinearODE.R", package = "dynr"))
```

1.1 Prepare the data

The first step in **dynr** modeling is to structure the data. This is done with the *dynr.data()* function.

```
> #-------------------------------------------------------------------------------
> # Example 1: Nonlinear Continuous-time Models
> #-------------------------------------------------------------------------------
> require(dynr)
> # ---- Read in the data ----
> data(PPsim)
> PPdata <- dynr.data(PPsim, id = "id", time = "time", observed = c("x", "y"))
```

The first argument of this function is either a *ts* class object of single-subject time series or a *data.frame* structured in a long (relational) format (i.e., with different measurement occasions from the same subject appearing as different rows in the data frame). Missing values in the observed variables should be indicated by *NA*. When a *ts* class object is passed to *dynr.data()*, no other inputs are needed. Otherwise, the *id* argument needs the name of the ID variable as input, and allows multiple people to be estimated in a single model by

distinguishing different individuals with the ID variable. That is, it indicates which rows should be modeled together as a time series. Thus, multi-subject modeling is as easy as single-subject modeling; only the data differ. The *time* argument needs the name of the TIME variable that indicates subject-specific measurement occasions. If a discrete-time model is desired, the TIME variable should contain subject-specific sequences of (subsets of) consecutively equally spaced numbers (e.g, 1, 2, 3, $\cdots$). In other words, the program assumes that the input *data.frame* is equally spaced with potential missingness. If the measurement occasions for a subject are a subset of an arithmetic sequence but are not consecutive, *NA*s will be inserted automatically to create an equally spaced data set before estimation. If a continuous-time model is being specified, the TIME variable can contain subject-specific increasing sequences of irregularly spaced real numbers. That is, the data may be input at their original, irregularly spaced intervals without the need to insert missingness. In this particular example, a discrete time model is used.

The *observed* and *covariates* arguments are used to indicate the names of the observed variables and covariates in the data. Covariates are defined as fixed predictors that are hypothesized to affect the modeling functions in one or more ways, but are otherwise not of interest (i.e., not modeled as dependent variables) to the user. Missing values in covariates are not allowed. That is, missing values in the covariates, if there are any, should be imputed first. The *dynr.data()* function lets users include data sets with many variables, but only use a few. The output of the function combines with the model recipe information later to map the model onto the data.

1.2 Prepare the recipes

The next step in **dynr** modeling is to build the recipes for the various parts of a model. The recipes are created with *prep.*()* functions.

1.2.1 Model specification: the dynamic functions

The dynamic model can take on the form of continuous-time models as

$$d\boldsymbol{\eta}_i(t) = \boldsymbol{f}_{S_i(t)}\left(\boldsymbol{\eta}_i(t), t, \boldsymbol{x}_i(t)\right) dt + d\boldsymbol{w}_i(t), \tag{3}$$

or the form of discrete-time state-space models [**Durbin01a**] as

$$\boldsymbol{\eta_i}(t_{i,j+1}) = \boldsymbol{f}_{S_i(t)}\left(\boldsymbol{\eta_i}(t_{i,j}), t_{i,j}, \boldsymbol{x}_i(t_{i,j})\right) + \boldsymbol{w}_i(t_{i,j+1}), \tag{4}$$

where i indexes person, t indexes time, $\boldsymbol{\eta}_i(t)$ is the $r \times 1$ vector of latent variables at time t, $\boldsymbol{x}_i(t)$ is the vector of covariates at time t, and $\boldsymbol{f}_{S_i(t)}(.)$ is the vector of (possibly nonlinear) dynamic functions. $\boldsymbol{f}_{S_i(t)}(.)$ depends on the latent regime indicator, $S_i(t)$, the discrete-valued latent variable that indexes the operating regime at time t.

The dynamic functions, $\boldsymbol{f}_{S_i(t)}()$ in Equations 3 and 4, can be specified using one of two possible functions in **dynr**: *prep.formulaDynamics()* and *prep.matrixDynamics()*.

While *prep.matrixDynamics()* can only be used for linear dynamic functions, *prep.formulaDynamics()* supports all native mathematical functions available in **R** and can be of use for both linear and nonlinear dynamic functions. Using the formula interface in **dynr**, the predator-prey model can be specified as:

```
> # dynamics
> preyFormula <- prey ~ a * prey - b * prey * predator
> predFormula <- predator ~ - c * predator + d * prey * predator
> ppFormula <- list(preyFormula, predFormula)
> ppDynamics <- prep.formulaDynamics(formula = ppFormula,
+   startval = c(a = 2.1, c = 0.8, b = 1.9, d = 1.1),
+   isContinuousTime = TRUE)
```

The first argument of the *prep.formulaDynamics()* function is *formula*. More specifically, this is a list of formulas. Each element in the list is a single, univariate, formula that defines a differential (if *isContinuousTime* = TRUE) or difference (if *isContinuousTime* = FALSE) equation. There should be one formula for every latent variable, in the order in which the latent variables are specified by using the *state.names* argument in *prep.measurement()*. The left-hand side of each formula is either the one-step-ahead projection of the latent variable (in the discrete-time case) or the differential of the latent variable (in the continuous-time case), namely, the left-hand-side of the respective dynamic equations. In both cases, users only need to specify the names of the latent variables that match the specification in *prep.measurement()* on the left-hand side of the formulas. The right-hand side of each formula gives a (linear or possibly nonlinear) function that may involve free or fixed parameters, numerical constants, exogenous covariates, and other arithmetic/mathematical functions that define the dynamics of the latent variables. The *startval* argument is a named vector giving the names of the free parameters and their starting values. Just as in the *prep.matrixDynamics()* function, the *isContinuousTime* argument is a binary flag that defines the switch between continuous- and discrete-time modeling.

With the formula interface, it is important to note that **dynr** uses the *D()* function from the **stats** package to automatically and symbolically differentiate the formulas provided. Hence, **dynr** uses the analytic Jacobian of the dynamics in its extended Kalman filter, greatly increasing its speed and accuracy. The *D()* function can handle the differentiation of functions involving parentheses, arithmetic operators (e.g., $+$, $-$, $*$, $/$, and $\hat{}$) and numerous mathematical functions (e.g., *exp*, *log*, *sin*, *cos*, *tan*, *sinh*, *cosh*, *sqrt*, *pnorm*, *dnorm*, *asin*, *acos*, *atan*, *gamma*, and so on). Thus, for a very large class of nonlinear functions, the user is spared from the need to supply the analytic Jacobian of the dynamic functions of interest to use the extended Kalman filter functionality in **dynr**. However, symbolic differentiation will not work for all formulas. For instance, formulas involving the absolute value function cannot be symbolically differentiated. For formulas that cannot be differentiated automatically using the **stats** package, the user must provide the analytic first derivatives through the *jacobian* argument. One can use the following code to find an example.

```
> file.edit(system.file("demo", "RSNonlinearDiscrete.R", package = "dynr"))
```

1.2.2 Model specification: the linear measurement function

We often assume that we have a simplest discrete-time measurement model in which $\boldsymbol{\eta}_i(t_{i,j})$ at discrete time point $t_{i,j}$ is indicated by a $r \times 1$ vector of manifest observations, $\boldsymbol{y}_i(t_{i,j})$ as

$$\boldsymbol{y}_i(t_{i,j}) = \boldsymbol{\eta}_i(t_{i,j}) + \boldsymbol{\epsilon}_i(t_{i,j}), \quad \boldsymbol{\epsilon}_i(t_{i,j}) \sim N\left(\mathbf{0}, \mathbf{R}_{S_i(t_{i,j})}\right), \tag{5}$$

which includes a $r \times 1$ vector of measurement errors $\boldsymbol{\epsilon}_i(t_{i,j})$ assumed to be serially uncorrelated over time and normally distributed with zero means and (possibly) regime-specific covariance matrix, $\mathbf{R}_{S_i(t_{i,j})}$. In this simplest case, the measurement model has the following two specifications.

```
> # Measurement (factor loadings)
> meas <- prep.measurement(
+   values.load = diag(1, 2),
+   obs.names = c('x', 'y'),
+   state.names = c('prey', 'predator'))
> # alternatively, use prep.loadings
> meas <- prep.loadings(
+   map = list(
+     prey = "x",
+     predator = "y"),
+   params = NULL)
```

1.2.3 Model specification: the latent and observed noise covariance structures

The noise recipe is created with *prep.noise()*. $\boldsymbol{w}_i(t)$ in Equation 3 is an r-dimensional Wiener process (i.e., continuous-time analog of a random walk process). The differentials of the Wiener processes have zero means and regime-specific covariance matrix, $\mathbf{Q}_{S_i(t)}$, often called the *diffusion* matrix. In Equation 4, however, $\boldsymbol{w}_i(t)$ denotes a vector of Gaussian distributed process noise with regime-specific covariance matrix, $\mathbf{Q}_{S_i(t)}$. In both continuous- and discrete-time models, $\mathbf{Q}_{S_i(t)}$ can be specified by the **.latent* arguments in *prep.noise()*. In ordinary differential equation models, $\mathbf{Q}_{S_i(t)} = \mathbf{0}$. The **.observed* arguments are used to specify $\mathbf{R}_{S_i(t_{i,j})}$ in Equation 5.

```
> #measurement and dynamics covariances
> mdcov <- prep.noise(
+   values.latent = diag(0, 2),
+   params.latent = diag(c("fixed", "fixed"), 2),
+   values.observed = diag(rep(0.3, 2)),
+   params.observed = diag(c("var_1", "var_2"), 2)
+ )
```

1.2.4 Model specification: the initial condition

In both the discrete- and continuous-time cases, the initial conditions for the dynamic functions are defined explicitly to be the latent variables at an individual-specific initial time point, $t_{i,1}$ (i.e., the first observed time point), denoted as $\boldsymbol{\eta}_i(t_{i,1})$, and are specified to be normally distributed with means $\boldsymbol{\mu}_{\boldsymbol{\eta}_1}$ and covariance matrix, $\boldsymbol{\Sigma}_{\boldsymbol{\eta}_1}$:

$$\boldsymbol{\eta}_i(t_{i,1}) \sim N\left(\boldsymbol{\mu}_{\boldsymbol{\eta}_1}, \boldsymbol{\Sigma}_{\boldsymbol{\eta}_1}\right). \tag{6}$$

```
> # Initial conditions on the latent state and covariance
> initial <- prep.initial(
+   values.inistate = c(3, 1),
+   params.inistate = c("fixed", "fixed"),
+   values.inicov = diag(c(0.01, 0.01)),
+   params.inicov = diag("fixed", 2)
+ )
```

1.2.5 Model specification: the transformation function

Many dynamic models may only lead to permissible (e.g., finite) values in particular parameter ranges. As such, we often need to add constraints to model parameters in fitting dynamic models. One way of doing this in **dynr** is to apply unconstrained optimization while transforming the parameters onto their constrained scales during function evaluations. This can be accomplished in **dynr** through the function *prep.tfun()*. For example, based on the nature of the predator and prey dynamics, the *a*-*d* parameters should, by right, take on positive values. Thus, we may choose to optimize their log-transformed values and exponentiate the unconstrained parameter values during likelihood evaluations to ensure that the values of these parameter estimates are always positive. To achieve this, we supply a list of transformation formulas to the *formula.trans* argument in the prep.tfun() function as follows:

```
> #constraints
> trans <- prep.tfun(formula.trans = list(a ~ exp(a), b ~ exp(b),
+                                         c ~ exp(c), d ~ exp(d)),
+   formula.inv = list(a ~ log(a), b ~ log(b),
+                      c ~ log(c), d ~ log(d)))
```

In cases involving the use of such constraint functions, the delta method is used to perform appropriate transformations to the covariance matrix of the parameter estimates at convergence to yield standard error estimates for the parameters on the constrained scales. If the starting values of certain parameters are indicated on a constrained scale, the *formula.inv* argument should give a list of inverse transformation formulas to transform the specified starting values to unconstrained scales for optimization.

1.3 Create and cook the model

After the recipes for all parts of the model are defined, the *dynr.model()* function creates the model and stores it in the *dynrModel* object. Each recipe (i.e., objects of class *dynrRecipe* created by *prep.*()*) and the data prepared by *dynr.data()* are given to this function. The function requires *dynamics*, *measurement*, *noise*, *initial*, and *data* as mandatory inputs for all models. When there are multiple regimes in the model, the *regimes* argument should be provided as shown below. When parameters are subject to transformation functions, a *transform* argument can be added. The *dynr.model()* function takes the recipes and the data and combines information from both. In doing so, this function uses the information from each recipe to write the text for a **C** function. Optionally, the **C** functions can be written to a file named by the *outfile* argument so that the user can inspect the automatically generated **C** code. Ideally of course, there is no need to ever examine this file; however, it is sometimes useful for debugging purposes and may be helpful for specifying models that extend those supported by the **R** interface functions. More frequently, inspecting the *dynrModel* object and "serving it" will provide the needed information.

```
> #-------------------------------------------------------------------------------
> # Cooking materials
>
> # Put all the recipes together in a Model Specification
> model2.1 <- dynr.model(dynamics = ppDynamics,
+   measurement = meas, noise = mdcov,
+   initial = initial, transform = trans,
+   data = PPdata,
+   outfile = "NonlinearODE.c")
> # Check the model specification
> printex(model2.1,
+   ParameterAs = model2.1$param.names,
+   show = FALSE, printInit = TRUE,
+   outFile = "NonlinearODE.tex")
> #tools::texi2pdf("NonlinearODE.tex")
> #system(paste(getOption("pdfviewer"), "NonlinearODE.pdf"))
>
> # Estimate free parameters
> res2.1 <- dynr.cook(dynrModel = model2.1)
>
```

In the last line above, the model is "cooked" with the *dynr.cook()* function to estimate the free parameters and their standard errors. When cooking, the **C** code that was written by *dynr.model()* is compiled and dynamically linked to the rest of the compiled **dynr** code. Then the **C** is executed to optimize the free parameters while calling the dynamically linked **C** functions that were created from the user-specified recipes. There are two points worth emphasizing in this regard. First, the user never has to write **C** functions. Second, the user benefits

from the **C** functions because of their speed. In this way, **dynr** provides an **R** interface for dynamical systems modeling while maintaining much of the speed associated with **C**.

1.4 Serve the results

The final step associated with **dynr** modeling is serving results (a *dynrCook* object) after the model has been cooked. To this end, several standard, popular S3 methods are defined for the *dynrCook* class, including *coef()*, *confint()*, *deviance()*, *logLik()* (and thus implicitly *AIC()* and *BIC()*), *names()*, *nobs()*, *summary()*, and *vcov()*. These methods perform the same tasks as their counterparts for regression models (i.e., *lm* class objects). The *summary()* method provides a table of free parameter names, estimates, standard errors, t-values, and Wald-type confidence intervals.

```
> # Examine results
> # True parameter values a = 2, b = 2, c = 1, d = 1
> summary(res2.1)
> #       names parameters       s.e.  t-value  ci.lower  ci.upper
> # a         a  1.9637320 0.06946322 28.27010 1.8275866 2.0998774
> # c         c  1.0023304 0.03062620 32.72788 0.9423042 1.0623567
> # b         b  1.9327832 0.06216237 31.09250 1.8109472 2.0546192
> # d         d  0.9608279 0.02628627 36.55246 0.9093078 1.0123481
> # var_1 var_1  0.2399578 0.01089095 22.03277 0.2186119 0.2613036
> # var_2 var_2  0.2380317 0.01072899 22.18585 0.2170033 0.2590601
> #
> # -2 log-likelihood value at convergence = 2843.19
> # AIC = 2855.19
> # BIC = 2884.64
```

2 Regime-switching extension

2.1 Prepare the data

```
> #-------------------------------------------------------------------------------
> # Example 2: Regime-Switching Nonlinear Continuous-time Model
> #-------------------------------------------------------------------------------
> # ---- Read in the data ----
> data("RSPPsim")
> data <- dynr.data(RSPPsim, id = "id", time = "time",
+   observed = c("x", "y"), covariate = "cond")
```

2.2 Prepare the recipes

Just as with the *prep.matrixDynamics()*, the formula interface also allows for regime-switching functionality. Consider an extension of the classical predator-

prey model. It is likely that the prey and predator interaction follow seasonal patterns. Hypothetically, we assume that in warmer seasons (i.e., "summer" environment), the interactions follow a classical predator-prey model, but in colder seasons (i.e., "winter" environment), the food source (e.g., grass) of the prey becomes limited and the predator species is able to find an additional food source due to the weather. So in the colder seasons the prey or predator population will not go to extreme values in absence of the other species. We thus consider the following regime-switching version of the predator-prey model to capture the potential seasonal changes in the interaction patterns. In the Summer regime, we have the predator-prey model as previously described, but in the Winter regime we now have a predator-prey model characterized by within-species competition and limiting growth/decay. In this competitive predator-prey model, the two populations do not grow/decline exponentially without bound in absence of the other, but rather, they grow logistically up to some finite carrying capacity. This logistic growth adds to the between-species interactions with the other population. This model can be specified by combining the predator and prey equations as:

```
> cPreyFormula <- prey ~ a * prey - e * prey ^ 2 - b * prey * predator
> cPredFormula <- predator ~ f * predator - c * predator ^ 2 + d * prey * predator
> cpFormula <- list(cPreyFormula, cPredFormula)
```

To specify the regime-switching predator-prey model, we combine the classical predator-prey model and the predator-prey model with within-species competition into a list of lists. Then we provide this list to the usual *prep.formulaDynamics()* function as the *formula* argument.

```
> rsFormula <- list(ppFormula, cpFormula)
> dynm <- prep.formulaDynamics(formula = rsFormula,
+    startval = c(a = 2.1, c = 3, b = 1.2, d = 1.2, e = 1, f = 2),
+    isContinuousTime = TRUE)
```

The phase portraits of the classical predator-prey model (Summer regime) and the cometitive predator-prey model (i.e., Winter regime) are shown in Figure 1 created by the **phaseR R** package [**phaseR**], where the two axes respectively represent the population size of the two species. In Figure 1(A), there is a reciprocal relation between the prey and predator population, whereas in Figure 1(B), there is an attractor or equilibrium state at $(1.5, 1.625)$, toward which the system tends to evolve.

2.2.1 Model specification: the regime-switching model

The initial class (or regime) probabilities for $S_i(t_{i,1})$ and the probabilities of transitions between regimes are represented using multinomial regression models as

$$\Pr\left(S_i(t_{i,1}) = m|\boldsymbol{x}_i(t_{i,1})\right) \quad \stackrel{\Delta}{=} \pi_{m,i1} = \frac{\exp(a_m + \mathbf{b}_m^T \boldsymbol{x}_i(t_{i,1}))}{\sum_{k=1}^{M} \exp(a_k + \mathbf{b}_k^T \boldsymbol{x}_i(t_{i,1}))}, \quad (7)$$

$$\Pr\left(S_i(t_{i,j}) = m|S_i(t_{i,j-1}) = l, \boldsymbol{x}_i(t_{i,j})\right) \quad \stackrel{\Delta}{=} \pi_{lm,it} = \frac{\exp(c_{lm} + \mathbf{d}_{lm}^T \boldsymbol{x}_i(t_{i,j})}{\sum_{k=1}^{M} \exp(c_{lk} + \mathbf{d}_{lk}^T \boldsymbol{x}_i(t_{i,j}))}, \quad (8)$$

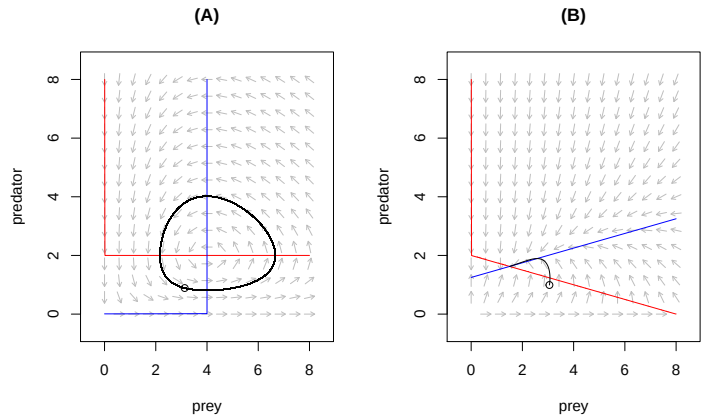

Figure 1: The phase portraits of (A) a classical predator-prey model and (B) a predator-prey model with within-species competition and limiting growth/decay.

where M denotes the total number of regimes, a_m denotes the logit intercept for the mth regime and $\mathbf{b}_m$ is a $n_b \times 1$ vector of regression slopes linked to a vector of covariates used to explain possible interindividual differences in initial log-odds (LO) of being in a regime relative to the reference regime selected by the user, operationalized as the regime where a_m and all entries in $\mathbf{b}_m$ are set to zero. Setting these entries to be zero in at least the reference regime is necessary for identification purposes: this ensures that the initial regime probabilities across all the hypothesized regimes sum to 1.0. $\pi_{lm,it}$ denotes individual i's probability of transitioning from class l at time $t_{i,j-1}$ to class m at time $t_{i,j}$ (i.e., the entry in the lth row and mth column of the transition probability matrix), c_{lm} denotes the logit intercept for the transition probability, and $\mathbf{d}_{lm}$ is a $n_d \times 1$ vector of logit slopes summarizing the effects of the covariates in $\boldsymbol{x}_i(t_{i,j})$ on that transition probability. The coefficients in $\mathbf{d}_{lm}$ are LO parameters representing the effects of the covariates on the LO of transitioning from the lth regime into the mth regime relative to transitioning into the reference regime - namely, the regime in which all LO parameters (including c_{lM} and all elements in $\mathbf{d}_{lM}^T$) are set to 0. One regime, again, has to be specified as the reference regime for identification purposes to ensure that conditional on being in a particular regime at time $t_{i,j-1}$, the probabilities of transitioning to each of the M regimes sum to 1.0 (i.e., $\sum_{m=1}^{M} \pi_{lm} = 1$).

The *prep.initial()* function is used to specify the model for the initial regime probabilities (i.e., Equation 7), in addition to the $\boldsymbol{\mu}_{\boldsymbol{\eta}_1}$ and $\boldsymbol{\Sigma}_{\boldsymbol{\eta}_1}$ in Equation 6. The *prep.regimes()* function specifies the structure of the regime switching functions shown in Equation 8. These two functions adopt similar structures for

specifying the parameters in the multinomial logistic regressions. Here we only focus on the specification of Equation 8 using the *prep.regimes()* function.

Note that based on Equation 8, a total of $n_d + 1$ parameters, including an intercept, c_{lm}, and n_d regression slopes in $\mathbf{d}_{lm}$, have to be defined for each of the functions governing the transition from the lth regime ($l = 1, \ldots, M$) to the mth regime ($m = 1, \ldots, M$). In total, there are $M \times M$ of such transition functions, corresponding to entries in an $M \times M$ transition probability matrix. The function *prep.regimes()* requires the user to provide the starting values (through the *values* argument) and names (through the *params* argument) for these $M \times (n_d + 1)$ parameters as a matrix whose number of rows equals to the number of regimes (i.e., M) and number of columns equals to the product of the number of regimes and the total number of parameters (i.e., $(n_d + 1)M$) as:

$$\left[\begin{array}{cc:cc:c:cc} c_{11} & \mathbf{d}_{11}^{\top} & c_{12} & \mathbf{d}_{12}^{\top} & \cdots & c_{1M} & \mathbf{d}_{1M}^{\top} \\ \hdashline c_{21} & \mathbf{d}_{21}^{\top} & c_{22} & \mathbf{d}_{22}^{\top} & \cdots & c_{2M} & \mathbf{d}_{2M}^{\top} \\ \hdashline \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hdashline c_{M1} & \mathbf{d}_{M1}^{\top} & c_{M2} & \mathbf{d}_{M2}^{\top} & \cdots & c_{MM} & \mathbf{d}_{MM}^{\top} \end{array}\right]. \tag{9}$$

In cases where covariates are involved in the multinomial logistic regression, a *covariates* argument allows us to provide the names of the covariates according to the order of the elements in $\mathbf{d_{lm}}$.

The example below shows equations and code that illustrate how covariates can be incorporated into the multinomial logistic regression (e.g., Equations 10 and 11 and neighboring blocks of code). In our hypothetical example, we have discussed how the weather condition may govern the regime switching processes. Specifically, we assume a covariate *cond* (with a value of 0 indicating the warmer weather and 1 indicating the colder weather) has an effect on the regime-switching transition probabilities. Then, we can specify the logistic regression model by

```
> # Regime-switching function
> # The RS model assumes that each element of the transition probability
> # matrix (TPM) can be expressed as a linear predictor (lp).
> # LPM =
> # lp(p11) ~ 1 + x1 + x2 + ... + xn,   lp(p12) ~ 1 + x1 + x2 + ... + xn
> # lp(p21) ~ 1 + x1 + x2 + ... + xn,   lp(p22) ~ 1 + x1 + x2 + ... + xn
> # Here I am specifying lp(p12) and lp(p22); the remaining elements
> # lp(p11) and lp(p21) are fixed at zero.
> # nrow = numRegimes, ncol = numRegimes*(numCovariates+1)
>
> regimes <- prep.regimes(
+   values = matrix(c(0, 0, -1, 1.5,
+                     0, 0, -1, 1.5),
+                   nrow = 2, ncol = 4, byrow = T),
+   params = matrix(c("fixed", "fixed", "int_1", "slp_1",
+                     "fixed", "fixed", "int_2", "slp_2"),
```

```
+                   nrow = 2, ncol = 4, byrow = T),
+   covariates = "cond")
```

In essence, the above code creates the following matrix in the form of

$$\left[\begin{array}{cc:cc} c_{11}=0 & d_{11}=0 & c_{12}=int_1=-1 & d_{12}=slp_1=1.5 \\ \hdashline c_{21}=0 & d_{21}=0 & c_{22}=int_2=-1 & d_{22}=slp_2=1.5 \end{array}\right], \tag{10}$$

which in turn creates the following transition probability matrix.

$$\begin{array}{c} \\ Summer_{t_{i,j}} \\ Winter_{t_{i,j}} \end{array} \begin{array}{c} \begin{array}{cc} Summer_{t_{i,j+1}} & Winter_{t_{i,j+1}} \end{array} \\ \begin{pmatrix} \frac{exp(0+0\times cond)}{exp(0+0\times cond)+exp(int_1+slp_1\times cond)} & \frac{exp(int_1+slp_1\times cond)}{exp(0+0\times cond)+exp(int_1+slp_1\times cond)} \\ \frac{exp(0+0\times cond)}{exp(0+0\times cond)+exp(int_2+slp_2\times cond)} & \frac{exp(int_2+slp_2\times cond)}{exp(0)+exp(int_2+slp_2\times cond)} \end{pmatrix} \end{array} \tag{11}$$

Here we consider the Summer regime as the reference regime, so the first two columns of the transition LO matrix (Equation 10) are fixed at zero. The third and fourth columns of the transition LO matrix respectively correspond to the regression intercepts and slopes associated with the covariate, whose starting values are respectively set at -1 and 1.5. With this set of starting values, the transition probability from any regime to the Summer regime is .73 when *cond* = 0, and .38 when *cond* = 1. The negative intercept implies that in warmer days (*cond* = 0), there is a greater chance of the process transitioning into the Summer regime, and the regression slope greater than the absolute value of the intercept suggests that in colder days (*cond* = 1), the transition into the Winter regime is more likely.

2.2.2 Model specification: the other components

A complete modeling script for this example can be retrieved using the code below.

```
> file.edit(system.file("demo", "RSNonlinearODE.R", package = "dynr"))
```

The rest of preparation before model fitting is shown below.

```
> # Measurement (factor loadings)
> meas <- prep.measurement(
+   values.load = diag(1, 2),
+   obs.names = c('x', 'y'),
+   state.names = c('prey', 'predator'))
> # Initial conditions on the latent state and covariance
> initial <- prep.initial(
+   values.inistate = c(3, 1),
+   params.inistate = c("fixed", "fixed"),
+   values.inicov = diag(c(0.01, 0.01)),
+   params.inicov = diag("fixed", 2),
+   values.regimep = c(.8473, 0),
```

```
+   params.regimep = c("fixed", "fixed"))
> #measurement and dynamics covariances
> mdcov <- prep.noise(
+   values.latent = diag(0, 2),
+   params.latent = diag(c("fixed","fixed"), 2),
+   values.observed = diag(rep(0.5,2)),
+   params.observed = diag(rep("var_epsilon",2),2)
+ )
> # dynamics
> preyFormula <- prey ~ a * prey - b * prey * predator
> predFormula <- predator ~ - c * predator + d * prey * predator
> ppFormula <- list(preyFormula, predFormula)
> cPreyFormula <- prey ~ a * prey - e * prey ^ 2 - b * prey * predator
> cPredFormula <- predator ~
+   f * predator - c * predator ^ 2 + d * prey * predator
> cpFormula <- list(cPreyFormula, cPredFormula)
> rsFormula <- list(ppFormula, cpFormula)
> dynm <- prep.formulaDynamics(formula = rsFormula,
+   startval = c(a = 2.1, c = 3, b = 1.2, d = 1.2, e = 1, f = 2),
+   isContinuousTime = TRUE)
> #constraints
> tformList <- list(a ~ exp(a), b ~ exp(b), c ~ exp(c),
+     d ~ exp(d), e ~ exp(e), f ~ exp(f))
> tformInvList <- list(a ~ log(a), b ~ log(b), c ~ log(c),
+     d ~ log(d), e ~ log(e), f ~ log(f))
> trans <- prep.tfun(
+     formula.trans = tformList,
+     formula.inv = tformInvList)
```

2.3 Create and cook the model

We fitted the specified model to the simulated data. In parameter estimation, *dynr* utilizes a sequential quadratic programming algorithm [**SLSQP1**, **SLSQP2**] available from an open-source library for nonlinear optimization — NLOPT [**NLopt**]. By default, we do not set boundaries on the parameters to be estimated. However, one can set the upper and lower boundaries of the estimated parameter values by respectively modifying the *ub* and *lb* slots of the model object of class *dynrModel*. An example is given as below to constrain the *int_1* and *int_2* parameters to be negative between -10 and 0, while limiting the values of *slp_1* and *slp_2* to positive within a range from 0 to 10. Similarly, the stopping criteria of the optimization algorithm can be modified through the *options* slot of the *dynrModel* object, which is a list consisting of specifications on the relative tolerance on optimization parameters (*xtol_rel*), the stopping threshold of the objective value (*stopval*), the absolute and relative tolerance on function value (i.e., *ftol_abs* and *ftol_rel*), the maximum number of function evaluations (*maxeval*), the maximum optimization time (in seconds; *maxtime*).

The output of the estimation function, *dynr.cook()*, is an object of class *dynrCook*. It not only includes estimation results that can be displayed in the summary table produced by *summary()*, but also contains information on posterior regime probabilities (i.e., the *pr_t_given_T* slot), smoothed state estimates of the latent variables (i.e., $\hat{\boldsymbol{\eta}}_i(t_{i,j}|T_i) = E(\boldsymbol{\eta}_i(t_{i,j})|\mathbf{Y}_i(T_i))$ in the *eta_smooth_final* slot), and smoothed error covariance matrices of the latent variables (i.e., $\mathbf{P}_i(t_{i,j}|T_i)$ in the *error_cov_smooth_final* slot) at all available time points. They can be retrieved by using the *$* operator.

```
> # Cooking materials
>
> # Put all the recipes together in a Model Specification
> model2.2 <- dynr.model(dynamics = dynm, measurement = meas,
+   noise = mdcov, initial = initial,
+   regimes = regimes, transform = trans,
+   data = data,
+   outfile = "RSNonlinearODE_1.c")
> # Check the model specification using LaTeX
> printex(model2.2, ParameterAs = names(model2.2), printInit = TRUE, printRS = TRUE,
+   outFile = "RSNonlinearODE_1.tex")
> #tools::texi2pdf("RSNonlinearODE_1.tex")
> #system(paste(getOption("pdfviewer"), "RSNonlinearODE_1.pdf"))
>
> model2.2$ub[ c("int_1", "int_2", "slp_1", "slp_2") ] <- c(0, 0, 10, 10)
> model2.2$lb[ c("int_1", "int_2", "slp_1", "slp_2") ] <- c(-10, -10, 0, 0)
> # Estimate free parameters
> res2.2 <- dynr.cook(model2.2)
> # Examine results
> summary(res2.2)
```

2.4 Serve the results

Figure 2 is created from the *dynr.ggplot()* (or *autoplot()*) method with *style* set to 2, and shows that the predicted trajectories match with the observed values and alternate between different regimes.

```
> dynr.ggplot(res2.2, model2.2, style = 2,
+   names.regime = c("Summer", "Winter"),
+   title = "", idtoPlot = 9,
+   text = element_text(size = 16))
```

Figure 3 is created by the *plotFormula()* method and presents the model equations with parameter names and estimated parameter values.

```
> plotFormula(model2.2, ParameterAs = names(model2.2)) +
+   ggtitle("(A)") +
+   theme(plot.title = element_text(hjust = 0.5, vjust = 0.01, size = 16))
```

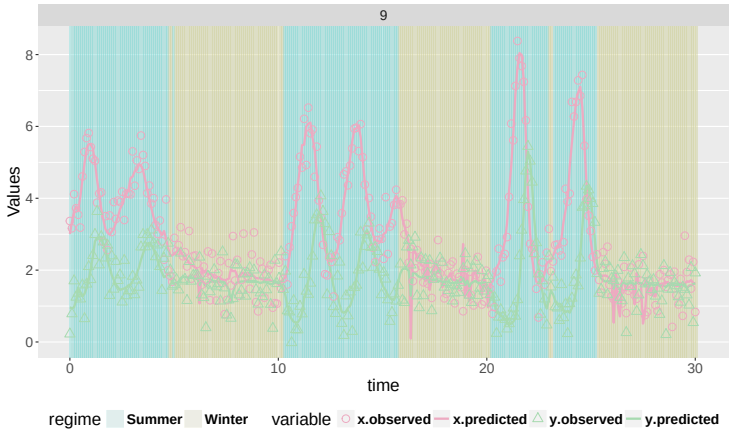

Figure 2: Built-in plotting feature for the predicted trajectories with observed values for the regime-switching nonlinear ODE model.

```
> plotFormula(model2.2, ParameterAs = coef(res2.2)) +
+   ggtitle("(B)") +
+   theme(plot.title = element_text(hjust = 0.5, vjust = 0.01, size = 16))
```

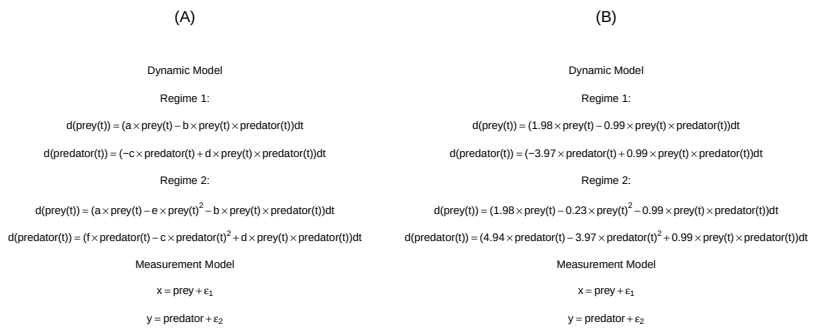

Figure 3: Automatic plots of model equations with (A) parameter names and (B) estimated parameters for the regime-switching nonlinear ODE model.