mvgam case study 1: model comparison and data assimilation

Nicholas Clark (n.clark@uq.edu.au)

Generalized Additive Models (GAMs) are flexible tools that have found particular application in analysis of time series. In ecology, a host of recent papers and workshops (i.e. the 2018 Ecological Society of America workshop on GAMs hosted by Eric Pedersen, David L. Miller, Gavin Simpson, and Noam Ross) have drawn attention to the power of GAMs for addressing complex ecological analyses. Given the many ways that GAMs can model temporal data, it is tempting to extrapolate from their smooth functions to produce out of sample forecasts. Here we will inspect the behaviours of smoothing splines when extrapolating outside the training data to examine whether this can be useful in practice.

We will work in the `mvgam` package, which fits Dynamic GAMs (DGAMs; Clark & Wells, 2022) using MCMC sampling (note that either `JAGS` or `Stan` is required; installation links are found here and here). Briefly, assume $\tilde{\boldsymbol{y}}_t$ is the conditional expectation of a discrete response variable $\boldsymbol{y}$ at time $\boldsymbol{t}$. Assuming $\boldsymbol{y}$ is drawn from an exponential distribution (such as Poisson or Negative Binomial) with a log link function, the linear predictor for a Dynamic GAM is written as:

$$log(\tilde{\boldsymbol{y}}_t) = \boldsymbol{B}_0 + \sum_{i=1}^{I} \boldsymbol{s}_{i,t}\boldsymbol{x}_{i,t} + \boldsymbol{z}_t \,,$$

Here $\boldsymbol{B}_0$ is the unknown intercept, the $\boldsymbol{s}$'s are unknown smooth functions of covariates ($\boldsymbol{x}$'s) and $\boldsymbol{z}$ is a dynamic latent trend. Each smooth function $\boldsymbol{s}_i$ is composed of basis expansions whose coefficients, which must be estimated, control the functional relationship between $\boldsymbol{x}_i$ and $log(\tilde{\boldsymbol{y}})$. The size of the basis expansion limits the smooth's potential complexity. A larger set of basis functions allows greater flexibility. Several advantages of GAMs are that they can model a diversity of response families, including discrete distributions (i.e. Poisson, Negative Binomial, Tweedie-Poisson) that accommodate common ecological features such as zero-inflation or overdispersion, and that they can be formulated to include hierarchical smoothing for multivariate responses. For the dynamic component, in its most basic form we assume a random walk with drift:

$$\boldsymbol{z}_t = \phi + \boldsymbol{z}_{t-1} + \boldsymbol{e}_t \,,$$

where ϕ is an optional drift parameter (if the latent trend is assumed to not be stationary) and $\boldsymbol{e}$ is drawn from a zero-centred Gaussian distribution. This model is easily modified to include autoregressive terms, which `mvgam` accomodates up to `order = 3`.

Why DGAMs?

Dynamic GAMs are useful when we wish to predict future values from time series that show temporal dependence and we want to avoid extrapolating a smooth (which, as you'll see below, can sometimes lead to unpredictable and unrealistic behaviours). In addition, smooths can often try to wiggle excessively to capture autocorrelation in a time series, which exacerbates the problem of forecasting ahead. Here we use an exaggerated example to show how a smooth tries to wiggle excessively, leading to overfitting by lessening the smoothing penalty λ.

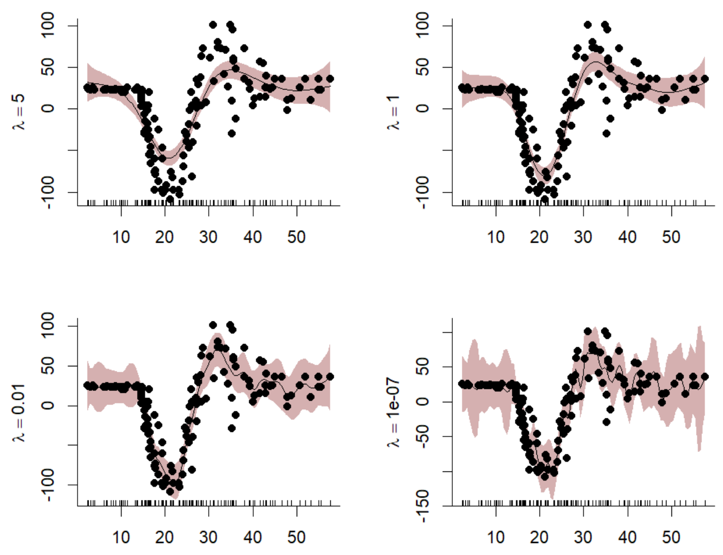
```
# Fit a model to the mcycle dataset in the MASS package, fixing the smoothing parameter at an abnormally
par(mfrow = c(2, 2),
    mgp = c(2.5, 1, 0),
    mai=c(c(0.7, 0.7, 0.2, 0.2)))
library(mgcv)
```

```
## Warning: package 'mgcv' was built under R version 4.2.2
```

```
## Loading required package: nlme
```

```
## This is mgcv 1.8-41. For overview type 'help("mgcv-package")'.
```

```r
data(mcycle, package = 'MASS')

m1 <- gam(accel ~ s(times, k = 50), data = mcycle, method = 'REML', sp = 5)

plot(m1, scheme = 1, residuals = TRUE, pch= 16,
     bty = 'L',
     ylab = expression(lambda == 5), xlab = '',
     shade.col = scales::alpha("#B97C7C", 0.6))

# Fit models with increasingly relaxed smooths (lambdas approaching zero)
m2 <- gam(accel ~ s(times, k = 50), data = mcycle, method = 'REML', sp = 1)
plot(m2, scheme = 1, residuals = TRUE, pch= 16,
     bty = 'L',
     ylab = expression(lambda == 1), xlab = '',
     shade.col = scales::alpha("#B97C7C", 0.6))

m3 <- gam(accel ~ s(times, k = 50), data = mcycle, method = 'REML', sp = 0.01)
plot(m3, scheme = 1, residuals = TRUE, pch= 16,
     bty = 'L',
     ylab = expression(lambda == 0.01), xlab = '',
     shade.col = scales::alpha("#B97C7C", 0.6))

m4 <- gam(accel ~ s(times, k = 50), data = mcycle, method = 'REML', sp = 0.0000001)
plot(m4, scheme = 1, residuals = TRUE, pch= 16,
     bty = 'L',
     ylab = expression(lambda == 0.0000001), xlab = '',
     shade.col = scales::alpha("#B97C7C", 0.6))
```



Notice how wiggly the function becomes in the last plot when λ is very small, indicating that the function is overfitting to the in-sample training data. Incidentally, this behaviour mirrors what `mgcv`'s `gam.check` function will often tell you to do if you are trying to model an autocorrelated time series with a smooth function. This is because the function will need a high degree of flexibility to model both the true function and the autocorrelation

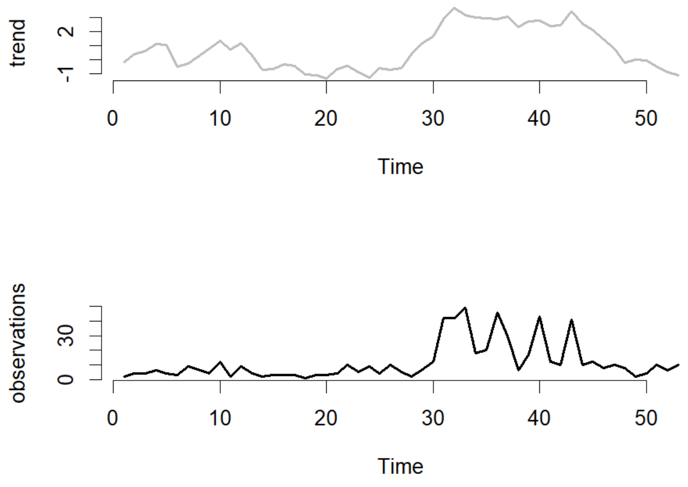
Simulation example

A simple simulation example is first used to demonstrate a comparison between a Dynamic GAM with a latent trend process vs a conventional GAM that includes an autoregressive observation model. First we simulate an autocorrelated trend and then draw noisy observations of that trend using a Negative Binomial observation process with overdispersion parameter = `10`

```r
#devtools::install_github("nicholasjclark/mvgam")
library(mvgam)
library(dplyr)
set.seed(1111)
trend <- cumsum(rnorm(53, -0.1, 0.5))
```

View the trend and the observation series. Here it is clear that there is autocorrelation in the trend and in the observations

```r
layout(matrix(1:2, ncol = 1, nrow = 2))
plot(trend, type = 'l', lwd = 2, col = 'grey', bty = 'none',
     xlab = 'Time')
observations <- rnbinom(53, size = 10, mu = 4 + exp(trend))
plot(observations, type = 'l', lwd = 2, bty = 'none',
     xlab = 'Time')
```



Gather the data into a dataframe format and calculate observation lags for lag = 1, 2, and 3

```r
data.frame(y = observations) %>%
  dplyr::mutate(lag1 = lag(y),
                lag2 = lag(y, 2),
                lag3 = lag(y, 3)) %>%
  dplyr::slice_tail(n = 50) -> sim_dat
sim_dat$time <- 1:50
```

Split the data into training and testing portions and examine the objects

```
data_train <- sim_dat[1:40,]
data_test <- sim_dat[41:50,]
head(data_train)
```

```
##   y lag1 lag2 lag3 time
## 1 6    4    4    2    1
## 2 4    6    4    4    2
## 3 3    4    6    4    3
## 4 9    3    4    6    4
## 5 7    9    3    4    5
## 6 4    7    9    3    6
```

```
head(data_test)
```

```
##     y lag1 lag2 lag3 time
## 41 10   41   10   12   41
## 42 12   10   41   10   42
## 43  8   12   10   41   43
## 44 10    8   12   10   44
## 45  8   10    8   12   45
## 46  2    8   10    8   46
```

As a first pass at modelling this series, we will fit a GAM with parametric effects of lags 1, 2, and 3. Our primary interest is to estimate the AR parameters and the overdispersion parameter of the Negative Binomial observation process

```
m_mgcv <- gam(y ~ lag1 + lag2 + lag3,
              data = data_train,
              method = "REML", family = nb())
```

We can view the summary of the `mgcv` model to see that the posterior estimates for the AR terms for lags 1 and 3 are strongly non-zero

```
summary(m_mgcv)
```

```
##
## Family: Negative Binomial(2.168)
## Link function: log
##
## Formula:
## y ~ lag1 + lag2 + lag3
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  1.61782    0.17936   9.020  < 2e-16 ***
## lag1         0.04014    0.01060   3.787 0.000153 ***
## lag2        -0.01193    0.01257  -0.949 0.342730
## lag3         0.03083    0.01053   2.928 0.003408 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##
## R-sq.(adj) =  0.00169   Deviance explained = 50.2%
## -REML =  142.1  Scale est. = 1         n = 40
```

Forecasting from this model requires a function to recursively update the design matrix (by filling in the values for the dynamic `lag(y)` covariates) at each successive one-step ahead prediction. Here we define a function to iteratively propagate a forecast forward according to the AR3 observation model. Note that we have used an argument to allow the lags to be fed into the model on the log scale, this will make more sense in a moment as we inspect the forecasts of the model

Click for definition of `recurse_ar3`

```r
recurse_ar3 = function(model, coef_sim, lagged_vals, h, log_lags = F){
  # Initiate state vector
  states <- rep(NA, length = h + 3)
  # Last three values of the conditional expectations begin the state vector
  if(log_lags){
    states[1] <- exp(as.numeric(lagged_vals[3]))
    states[2] <- exp(as.numeric(lagged_vals[2]))
    states[3] <- exp(as.numeric(lagged_vals[1]))
  } else {
    states[1] <- as.numeric(lagged_vals[3])
    states[2] <- as.numeric(lagged_vals[2])
    states[3] <- as.numeric(lagged_vals[1])
  }

  # Get a random sample of the smooth coefficient uncertainty matrix
  # to use for the entire forecast horizon of this particular path
  gam_coef_index <- sample(seq(1, NROW(coef_sim)), size = 1)
  # For each following timestep, recursively predict based on the
  # predictions at each previous lag
  for (t in 4:(h + 3)) {
    newdata <- data_test[t-3, ]
    if(log_lags){
      newdata$lag1 <- log(states[t-1] + 0.01)
      newdata$lag2 <- log(states[t-2] + 0.01)
      newdata$lag3 <- log(states[t-3] + 0.01)
    } else {
      newdata$lag1 <- states[t-1]
      newdata$lag2 <- states[t-2]
      newdata$lag3 <- states[t-3]
    }

    colnames(newdata) <- colnames(data_test)
    Xp <- predict(model, newdata = newdata, type = 'lpmatrix')
    # Calculate the posterior prediction for this timepoint; use an upper constraint
    # to ensure the values don't blow up to infinity
    mu <- rnbinom(1, mu = min(100000, exp(Xp %*% coef_sim[gam_coef_index,])),
                  size = model$family$getTheta(TRUE))
    # Fill in the state vector and iterate to the next timepoint
    states[t] <- mu
  }
  # Return the forecast path
  states[-c(1:3)]
}
```

Draw 1000 parameter estimates from the GAM posterior and use these to propagate the forecast for 10 timesteps ahead

```r
coef_sim <- MASS::mvrnorm(1000, mu = m_mgcv$coefficients, Sigma = m_mgcv$Vp)
gam_sims <- matrix(NA, nrow = 1000, ncol = 10)
for(i in 1:1000){
  gam_sims[i,] <- recurse_ar3(m_mgcv,
                              coef_sim,
                              lagged_vals = c(data_test$lag1[1],
                                              data_test$lag2[1],
                                              data_test$lag3[1]),
                              h = 10)
}
```

Now we will fit a competing Dynamic GAM model, which estimates a latent temporal process rather than using an autoregressive observation model. Note that to actually condition models with MCMC sampling, either the `JAGS` software must be installed (along with the `R` packages `rjags` and `runjags`) or the `Stan` software must be installed (along with the package `rstan` and, optionally, the `cmdstanr` package). These are not listed as dependencies of `mvgam` to ensure that installation is less difficult. If users wish to fit the models using `mvgam`, please refer to installation links for `JAGS` here or for `Stan` and `rstan` here). This model's parameters are estimated in a Bayesian framework using (by default) the Gibbs sampling algorithms available in the `JAGS` software. `mvgam` takes a fitted `gam` model and adapts the model file to fit in `JAGS`, with possible extensions to deal with stochastic trend components and other features.

```
m_mvgam <- mvgam(formula = y ~ 1,
                 data = data_train,
                 newdata = data_test,
                 family = 'nb',
                 trend_model = 'AR3',
                 chains = 4,
                 burnin = 1000)
```

You can view the summary of the resulting model using the `summary.mvgam S3` function, which gives an overview of convergence and effective sample size diagnostics for the model's parameters

```
summary(m_mvgam)
```

```
## GAM formula:
## y ~ 1
##
## Family:
## Negative Binomial
##
## Link function:
## log
##
## Trend model:
## AR3
##
## N series:
## 1
##
## N observations:
```

```
## 40

##

## Status:

## Fitted using JAGS

##

## Dispersion parameter estimates:

##       2.5%      50%    97.5% Rhat n.eff
## r 3.049277 10.50643 287.9789 1.17  1720

##

## GAM coefficient (beta) estimates:

##                 2.5%      50%    97.5% Rhat n.eff
## (Intercept) 1.478364 2.285138 2.909908 1.27    32

##

## Latent trend parameter estimates:

##             2.5%       50%     97.5% Rhat n.eff
## ar1    0.1012778 0.4573410 0.8324768 1.03   655
## ar2   -0.2755530 0.1546750 0.5768053 1.00   426
## ar3   -0.1354848 0.2496544 0.6040130 1.03   614
## sigma  0.2995692 0.5774133 0.8470327 1.07   165

##

## JAGS MCMC diagnostics

## Rhats above 1.05 found for 53 parameters
## *Diagnose further to investigate why the chains have not mixed

##
```

The model file can also be viewed, which can be handy for making modifications to the model so that extra stochastic elements can be added to a user's bespoke model. Notice how the summary at the top of the model file describes the data elements (and their dimensions) that are needed to condition the model, again the hope is that this simplifies the task of modifying the model for any additional user-specific requirements

```
m_mvgam$model_file
```

```
##  [1] "JAGS model code generated by package mvgam"
##  [2] ""
##  [3] "GAM formula:"
##  [4] "y ~ 1"
##  [5] ""
##  [6] "Trend model:"
##  [7] "AR3"
##  [8] ""
##  [9] "Required data:"
## [10] "integer n;  number of timepoints per series"
## [11] "integer n_series;  number of series"
## [12] "matrix y;  time-ordered observations of dimension n x n_series (missing values allowed)"
## [13] "matrix ytimes;  time-ordered n x n_series matrix (which row in X belongs to each [time, series
## [14] "matrix X;  mgcv GAM design matrix of dimension (n x n_series) x basis dimension"
## [15] "vector p_coefs;  vector (length = 1) of prior Gaussian means for parametric effects"
## [16] "vector p_taus;  vector (length = 1) of prior Gaussian precisions for parametric effects"
## [17] "min_eps; .Machine$double.eps (smallest floating-point number x such that 1 + x != 1)"
## [18] ""
## [19] ""
## [20] "#### Begin model ####"
## [21] "model {"
## [22] ""
## [23] "## GAM linear predictor"
## [24] "eta <- X * b"
## [25] ""
## [26] "## mean expectations"
## [27] "for (i in 1:n) {"
## [28] "for (s in 1:n_series) {"
## [29] "mus[i, s] <- exp(eta[ytimes[i, s]] + trend[i, s])"
## [30] "}"
## [31] "}"
## [32] ""
## [33] "## trend estimates"
## [34] "for (s in 1:n_series) {"
## [35] "trend[1, s] ~ dnorm(0, tau[s])"
## [36] "}"
## [37] ""
## [38] "for (s in 1:n_series) {"
## [39] "trend[2, s] ~ dnorm(ar1[s]*trend[1, s], tau[s])"
## [40] "}"
## [41] ""
## [42] "for (s in 1:n_series) {"
## [43] "trend[3, s] ~ dnorm(ar1[s]*trend[2, s] + ar2[s]*trend[1, s], tau[s])"
## [44] "}"
## [45] ""
## [46] "for (i in 4:n) {"
## [47] "for (s in 1:n_series){"
## [48] "trend[i, s] ~ dnorm(ar1[s]*trend[i - 1, s] + ar2[s]*trend[i - 2, s] + ar3[s]*trend[i - 3, s], 
## [49] "}"
## [50] "}"
## [51] ""
## [52] "## AR components"
## [53] "for (s in 1:n_series){"
## [54] "ar1[s] ~ dnorm(0, 10)"
## [55] "ar2[s] ~ dnorm(0, 10)"
## [56] "ar3[s] ~ dnorm(0, 10)"
## [57] "tau[s] <- pow(sigma[s], -2)"
## [58] "sigma[s] ~ dexp(2)T(0.075, 5)"
## [59] "}"
## [60] ""
## [61] "## likelihood functions"
## [62] "for (i in 1:n) {"
## [63] "for (s in 1:n_series) {"
## [64] "y[i, s] ~ dnegbin(rate[i, s], r[s])"
## [65] "rate[i, s] <- ifelse((r[s] / (r[s] + mus[i, s])) < min_eps, min_eps,"
## [66] "(r[s] / (r[s] + mus[i, s])))"
## [67] "}"
## [68] "}"
## [69] ""
## [70] "## complexity penalising prior for the overdispersion parameter;"
## [71] "## where the likelihood reduces to a 'base' model (Poisson) unless"
## [72] "## the data support overdispersion"
## [73] "for (s in 1:n_series) {"
## [74] "r[s] <- 1 / r_inv[s]"
## [75] "r_inv[s] ~ dexp(5)"
## [76] "}"
## [77] ""
## [78] "## posterior predictions"
## [79] "for (i in 1:n) {"
## [80] "for (s in 1:n_series) {"
## [81] "ypred[i, s] ~ dnegbin(rate[i, s], r[s])"
## [82] "}"
## [83] "}"
## [84] ""
## [85] "## GAM-specific priors"
## [86] "## parametric effect priors (regularised for identifiability)"
## [87] "for (i in 1:1) { b[i] ~ dnorm(p_coefs[i], p_taus[i]) }"
## [88] "}"
```

Users can also view the data and initial values that `mvgam` uses to condition the model by either setting the argument `return_model_data = TRUE` or by setting `run_model = FALSE`. The latter is faster for simply generating the necessary objects and initial value functions. These options will also return the `pregam` structure that is necessary to convert `MCMC` samples into an object that can be readily used by `mgcv` to generate predictions and calculate important quantities such as effective degrees of freedom (see `help(jagam)` for more details)

```
mod_skeleton <- mvgam(y ~ 1,
                 data = data_train,
                 newdata = data_test,
                 family = 'nb',
                 trend_model = 'AR3',
                 chains = 4,
                 burnin = 1000,
                 run_model = FALSE)
```

```
## Warning in newton(lsp = lsp, X = G$X, y = G$y, Eb = G$Eb, UrS = G$UrS, L =
## G$L, : gam.fit3 algorithm did not converge
```

```
str(mod_skeleton$model_data)
```

```
## List of 8
##  $ y       : num [1:50, 1] 6 4 3 9 7 4 12 2 9 4 ...
##  $ n       : int 50
##  $ X       : num [1:50, 1] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:50] "X" "X.1" "X.2" "X.3" ...
##   .. ..$ : chr "X.Intercept."
##  $ p_coefs : Named num 2.57
##   ..- attr(*, "names")= chr "(Intercept)"
##  $ p_taus  : num 4.59
##  $ ytimes  : int [1:50, 1] 1 2 3 4 5 6 7 8 9 10 ...
##  $ n_series: int 1
##  $ min_eps : num 2.22e-16
```

```r
mod_skeleton$inits
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
```

```r
head(mod_skeleton$pregam)
```

```
## $m
## [1] 1
##
## $min.sp
## NULL
##
## $pearson.extra
## [1] 0
##
## $dev.extra
## [1] 0
##
## $n.true
## [1] -1
##
## $pterms
## y ~ 1
## attr(,"variables")
## list(y)
## attr(,"factors")
## integer(0)
## attr(,"term.labels")
## character(0)
## attr(,"order")
## integer(0)
## attr(,"intercept")
## [1] 1
## attr(,"response")
## [1] 1
## attr(,".Environment")
## <environment: R_GlobalEnv>
## attr(,"predvars")
## list(y)
## attr(,"dataClasses")
##         y
## "numeric"
```

The same steps above can be used to generate `Stan` data and modelling files:

```r
mod_skeleton <- mvgam(y ~ 1,
                      data = data_train,
                      newdata = data_test,
                      family = 'nb',
                      trend_model = 'AR3',
                      chains = 4,
                      use_stan = TRUE,
                      run_model = FALSE)
```

```
## Warning in newton(lsp = lsp, X = G$X, y = G$y, Eb = G$Eb, UrS = G$UrS, L =
## G$L, : gam.fit3 algorithm did not converge
```

```r
str(mod_skeleton$model_data)
```

```
## List of 15
##  $ y           : num [1:50, 1] 6 4 3 9 7 4 12 2 9 4 ...
##  $ n           : int 50
##  $ X           : num [1:50, 1] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:50] "X" "X.1" "X.2" "X.3" ...
##   .. ..$ : chr "X.Intercept."
##  $ p_coefs     : num [1(1d)] 2.57
##  $ p_taus      : num [1(1d)] 4.73
##  $ ytimes      : int [1:50, 1] 1 2 3 4 5 6 7 8 9 10 ...
##  $ n_series    : int 1
##  $ min_eps     : num 2.22e-16
##  $ y_observed  : num [1:50, 1] 1 1 1 1 1 1 1 1 1 1 ...
##  $ total_obs   : int 50
##  $ num_basis   : int 1
##  $ n_nonmissing: int 40
##  $ obs_ind     : int [1:40] 1 2 3 4 5 6 7 8 9 10 ...
##  $ flat_ys     : num [1:40] 6 4 3 9 7 4 12 2 9 4 ...
##  $ flat_xs     : num [1:40, 1] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:40] "X" "X.1" "X.2" "X.3" ...
##   .. ..$ : NULL
```

```r
mod_skeleton$inits
```

```
## function() {
##                 list(b_raw = runif(model_data$num_basis, -2, 2),
##                      r_inv = runif(NCOL(model_data$ytimes), 1, 50),
##                      sigma = runif(model_data$n_series, 0.075, 1))
##               }
## <bytecode: 0x00000202ef811b40>
## <environment: 0x00000202e654fae0>
```

```r
head(mod_skeleton$pregam)
```

```
## $m
## [1] 1
##
## $min.sp
## NULL
##
## $pearson.extra
## [1] 0
##
## $dev.extra
## [1] 0
##
## $n.true
## [1] -1
##
## $pterms
## y ~ 1
## attr(,"variables")
## list(y)
## attr(,"factors")
## integer(0)
## attr(,"term.labels")
## character(0)
## attr(,"order")
## integer(0)
## attr(,"intercept")
## [1] 1
## attr(,"response")
## [1] 1
## attr(,".Environment")
## <environment: R_GlobalEnv>
## attr(,"predvars")
## list(y)
## attr(,"dataClasses")
##         y
## "numeric"
```

The Stan model file is also designed to be as descriptive as possible about the data objects that are required to condition the model

```
cat(c("```stan", mod_skeleton$model_file, "```"), sep = "\n")
```

```stan
// Stan model code generated by package mvgam
functions {
vector rep_each(vector x, int K) {
int N = rows(x);
vector[N * K] y;
int pos = 1;
for (n in 1:N) {
for (k in 1:K) {
y[pos] = x[n];
pos += 1;
}
}
return y;
}
}
data {
int<lower=0> total_obs; // total number of observations
int<lower=0> n; // number of timepoints per series
int<lower=0> n_series; // number of series
int<lower=0> num_basis; // total number of basis coefficients
```

```
real p_taus[1]; // prior precisions for parametric coefficients
real p_coefs[1]; // prior locations for parametric coefficients
matrix[total_obs, num_basis] X; // mgcv GAM design matrix
int<lower=0> ytimes[n, n_series]; // time-ordered matrix (which col in X belongs to each [time, series]

int<lower=0> n_nonmissing; // number of nonmissing observations
int<lower=0> flat_ys[n_nonmissing]; // flattened nonmissing observations
matrix[n_nonmissing, num_basis] flat_xs; // X values for nonmissing observations
int<lower=0> obs_ind[n_nonmissing]; // indices of nonmissing observations
}
parameters {
// raw basis coefficients
vector[num_basis] b_raw;

// negative binomial overdispersion
vector<lower=0>[n_series] r_inv;

// latent trend AR1 terms
vector<lower=-1.5,upper=1.5>[n_series] ar1;

// latent trend AR2 terms
vector<lower=-1.5,upper=1.5>[n_series] ar2;

// latent trend AR3 terms
vector<lower=-1.5,upper=1.5>[n_series] ar3;

// latent trend variance parameters
vector<lower=0>[n_series] sigma;

// latent trends
matrix[n, n_series] trend;

}

transformed parameters {
// basis coefficients
vector[num_basis] b;

b[1:num_basis] = b_raw[1:num_basis];

}

model {
for (i in 1:1) {
b_raw[i] ~ normal(p_coefs[i], 1 / p_taus[i]);
}

// priors for AR parameters
ar1 ~ std_normal();
ar2 ~ std_normal();
ar3 ~ std_normal();

// priors for overdispersion parameters
```

```
r_inv ~ student_t(3, 0, 0.1);

// priors for latent trend variance parameters
sigma ~ exponential(2);

// trend estimates
trend[1, 1:n_series] ~ normal(0, sigma);
trend[2, 1:n_series] ~ normal(trend[1, 1:n_series] * ar1, sigma);
trend[3, 1:n_series] ~ normal(trend[2, 1:n_series] * ar1 + trend[1, 1:n_series] * ar2, sigma);
for(s in 1:n_series){
trend[4:n, s] ~ normal(ar1[s] * trend[3:(n - 1), s] + ar2[s] * trend[2:(n - 2), s] + ar3[s] * trend[1:(
}

{
// likelihood functions
vector[n_nonmissing] flat_trends;
real flat_rs[n_nonmissing];
flat_trends = (to_vector(trend))[obs_ind];
flat_rs = to_array_1d(rep_each(r_inv, n)[obs_ind]);
flat_ys ~ neg_binomial_2(
exp(append_col(flat_xs, flat_trends) * append_row(b, 1.0)),
inv(flat_rs));
}
}


generated quantities {
vector[total_obs] eta;
matrix[n, n_series] mus;
vector[n_series] tau;
array[n, n_series] int ypred;
matrix[n, n_series] r_vec;
vector[n_series] r;
r = inv(r_inv);
for (s in 1:n_series) {
r_vec[1:n,s] = rep_vector(r[s], n);
}

for (s in 1:n_series) {
tau[s] = pow(sigma[s], -2.0);
}

// posterior predictions
eta = X * b;
for(s in 1:n_series){
mus[1:n, s] = eta[ytimes[1:n, s]] + trend[1:n, s];
ypred[1:n, s] = neg_binomial_2_rng(exp(mus[1:n, s]), r_vec[1:n, s]);
}
}
```

Now we will evaluate the forecasts for each of the models. Here we define a function to plot posterior empirical forecast quantiles and calculate the Discrete Rank Probability Score (DRPS, a proper scoring rule for evaluating probabilistic forecast distributions) for each of the out of sample forecasts. The final score for each forecast is the sum of DRPS values at each forecast horizon, with the lower score being the favoured forecast. The function returns a plot of the forecast distribution and also prints the DRPS score

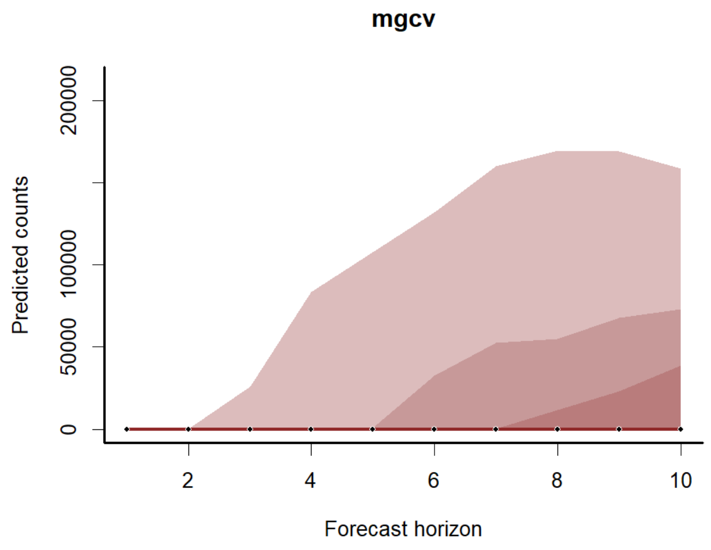
Click for definition of `plot_fc_horizon`

```r
plot_fc_horizon = function(gam_sims, main){
  probs = c(0.05, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.95)
  cred <- sapply(1:NCOL(gam_sims),
                 function(n) quantile(gam_sims[,n],
                                      probs = probs))

  c_light <- c("#DCBCBC")
  c_light_highlight <- c("#C79999")
  c_mid <- c("#B97C7C")
  c_mid_highlight <- c("#A25050")
  c_dark <- c("#8F2727")
  c_dark_highlight <- c("#7C0000")

  ylim <- range(c(0, data_test$y[1:NCOL(cred)] * 1.25,
                  cred * 1.25))
  pred_vals <- 1:NCOL(cred)
  plot(1, xlim = c(1, NCOL(cred)),
       bty = 'L',
       col = rgb(1,0,0, alpha = 0),
       ylim = ylim,
       ylab = 'Predicted counts',
       xlab = 'Forecast horizon',
       main = main)

  polygon(c(pred_vals, rev(pred_vals)), c(cred[1,], rev(cred[9,])),
          col = c_light, border = NA)
  polygon(c(pred_vals, rev(pred_vals)), c(cred[2,], rev(cred[8,])),
          col = c_light_highlight, border = NA)
  polygon(c(pred_vals, rev(pred_vals)), c(cred[3,], rev(cred[7,])),
          col = c_mid, border = NA)
  polygon(c(pred_vals, rev(pred_vals)), c(cred[4,], rev(cred[6,])),
          col = c_mid_highlight, border = NA)
  lines(pred_vals, cred[5,], col = c_dark, lwd = 2.5)
  points(data_test$y[1:NCOL(cred)], pch = 16, cex = .65, col = 'white')
  points(data_test$y[1:NCOL(cred)], pch = 16, cex = .55)
  box(bty = 'L', lwd = 2)

  # Calculate out of sample DRPS and print the score
  drps_score <- function(truth, fc, interval_width = 0.9){
    nsum <- 1000
    Fy = ecdf(fc)
    ysum <- 0:nsum
    indicator <- ifelse(ysum - truth >= 0, 1, 0)
    score <- sum((indicator - Fy(ysum))^2)

    # Is value within empirical interval?
    interval <- quantile(fc, probs = c((1-interval_width)/2,
                                       (interval_width + (1-interval_width)/2)))
    in_interval <- ifelse(truth <= interval[2] & truth >= interval[1], 1, 0)
    return(c(score, in_interval))
  }
  # Wrapper to operate on all observations in fc_horizon
  drps_mcmc_object <- function(truth, fc, interval_width = 0.9){
    indices_keep <- which(!is.na(truth))
    if(length(indices_keep) == 0){
      scores = data.frame('drps' = rep(NA, length(truth)),
                          'interval' = rep(NA, length(truth)))
    } else {
      scores <- matrix(NA, nrow = length(truth), ncol = 2)
      for(i in indices_keep){
        scores[i,] <- drps_score(truth = as.vector(truth)[i],
                                 fc = fc[,i], interval_width)
      }
    }
    scores
  }

  truth <- data_test$y[1:NCOL(cred)]
  fc <- gam_sims

  message('Out of sample DRPS:')
  print(sum(drps_mcmc_object(as.vector(truth),
                             fc)[,1]))
  message()

}
```

Evaluate forecasts for the two models; first the `mgcv` autoregressive observation model

```
plot_fc_horizon(gam_sims = gam_sims[,1:10], main = 'mgcv')
```

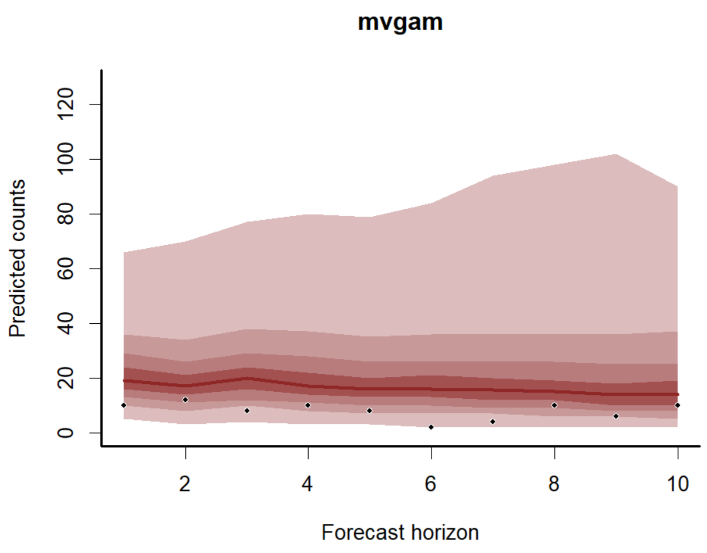
```
## Out of sample DRPS:
## [1] 649.8801
##
```



And now the `mvgam` latent trend model

```
plot_fc_horizon(gam_sims = MCMCvis::MCMCchains(m_mvgam$model_output,
                                               'ypred')[,(NROW(m_mvgam$obs_data)+1):
                                                          (NROW(m_mvgam$obs_data)+10)],
                main = 'mvgam')
```

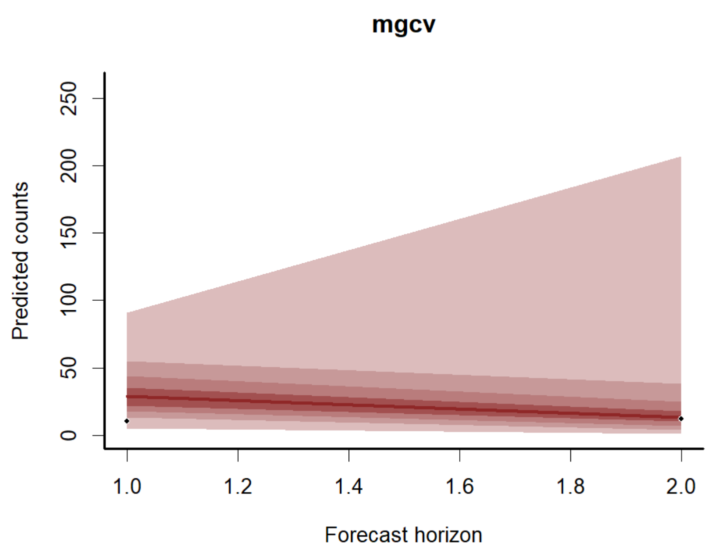
```
## Out of sample DRPS:

## [1] 66.07028

##
```



Generally the uncertainy blows out when forecasting further ahead for the `mgcv` model (why might this be?); if we only look 2 timesteps ahead, it is not quite as terrible but it becomes more clear what is happening

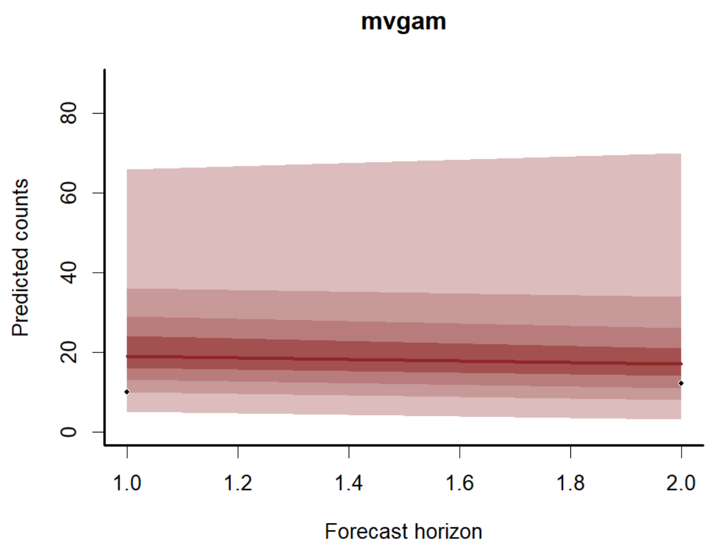
```
plot_fc_horizon(gam_sims = gam_sims[,1:2], main = 'mgcv')
```

```
## Out of sample DRPS:

## [1] 18.44607

##
```



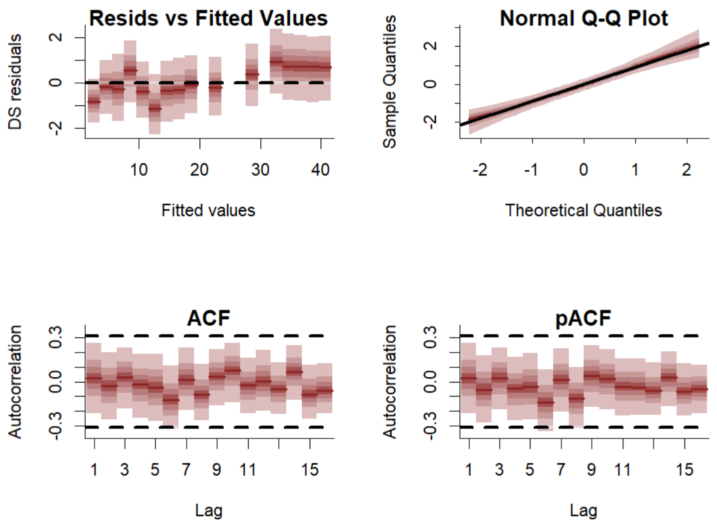
```r
plot_fc_horizon(gam_sims = MCMCvis::MCMCchains(m_mvgam$model_output,
                                               'ypred')[,(NROW(m_mvgam$obs_data)+1):
                                                          (NROW(m_mvgam$obs_data)+2)],
                main = 'mvgam')
```

```
## Out of sample DRPS:
## [1] 10.8586
##
```

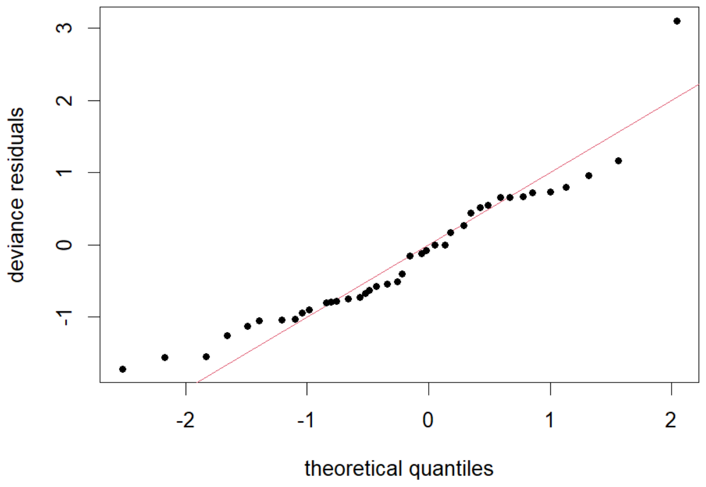


Why does autocorrelated observation model perform so badly? Q-Q plots give some insight, as they clearly indicate the dynamic latent trend model shows a better fit of the observation process to the data compared to the autocorrelated observation model

```r
plot(m_mvgam, 'residuals')
```



```
qq.gam(m_mgcv, pch = 16, lwd = 3, cex = 0.8)
```



In fact the the autocorrelated observation model is overestimating the amount of overdispersion in the data. This can partially explain the exponentially growing uncertainties in the mgcv forecast The truth for the ovderdispersion parameter is `10`. This is likely driven by the autoregressive model's high sensitivity to the noisy observation process. Observations in ecology very often display properties of overdispersion, and this results in observation 'outliers' that can have large leverage on resulting parameter estimates for autoregressive models. The noisy observation process is not a problem for latent trend models such as those estimated by **`mvgam`**, as the observation model is estimated separately from the temporal process (as is the norm for state-space models)

```
m_mgcv$family$getTheta(TRUE)
```

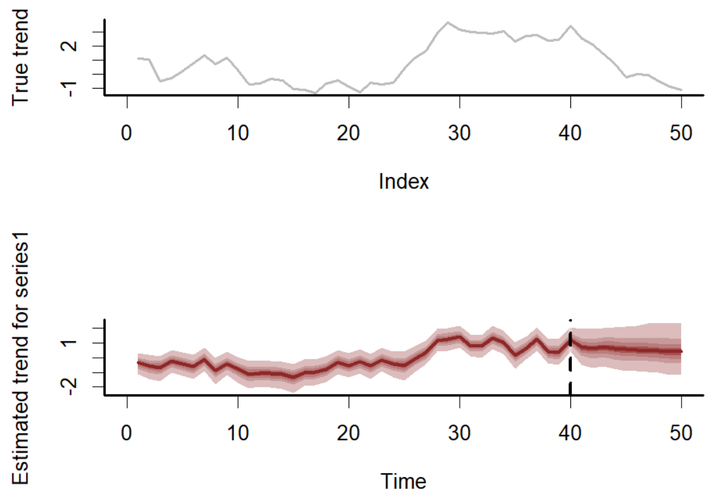
```
## [1] 2.167732
```

```
quantile(MCMCvis::MCMCchains(m_mvgam$model_output, 'r'), c(0.1, 0.5, 0.9))
```

```
##       10%       50%       90%
##  4.17315 10.50643 67.28672
```

Looking at estimates for the full trend and forecast distributions shows how the `mvgam` model is clearly doing a decent job of tracking the latent state up to the end of the training period while also giving reasonable estimates for the overdispersion in the data

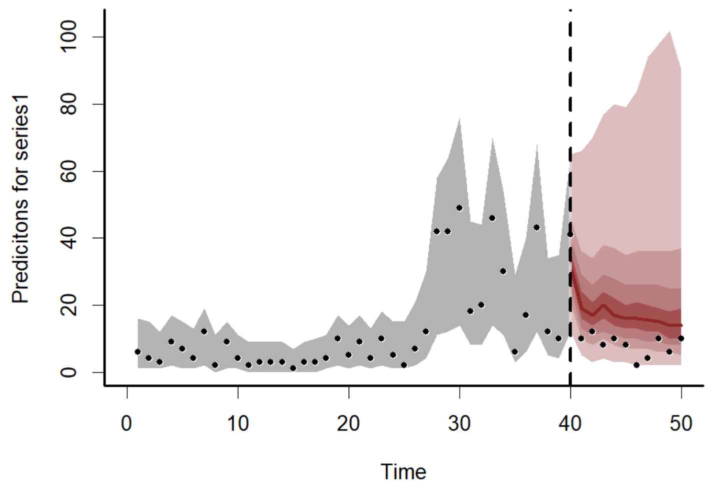
```
layout(matrix(1:2, ncol = 1, nrow = 2))
plot(trend[4:53], xlim = c(0, 50), bty = 'L', type = 'l', lwd = 2, col = 'grey',
     ylab = 'True trend')
box(bty = 'L', lwd = 2)
plot(m_mvgam, 'trend', newdata = data_test)
```



```
layout(1)
```

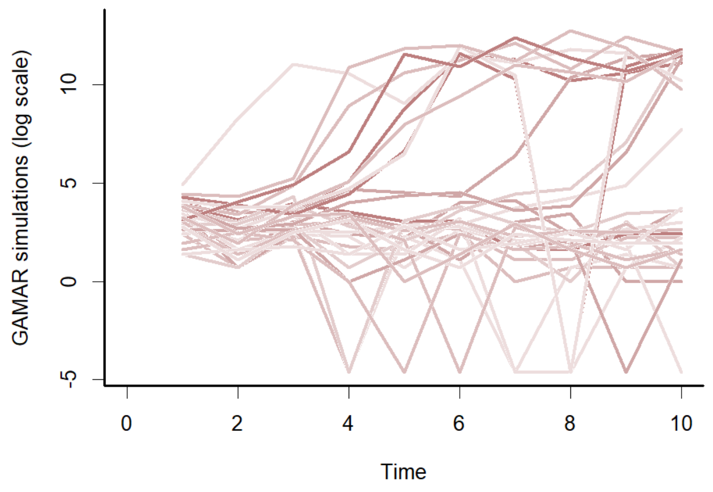
```
plot(m_mvgam, 'forecast', newdata = data_test)
```

```
## Out of sample DRPS:

## [1] 66.07028

##
```



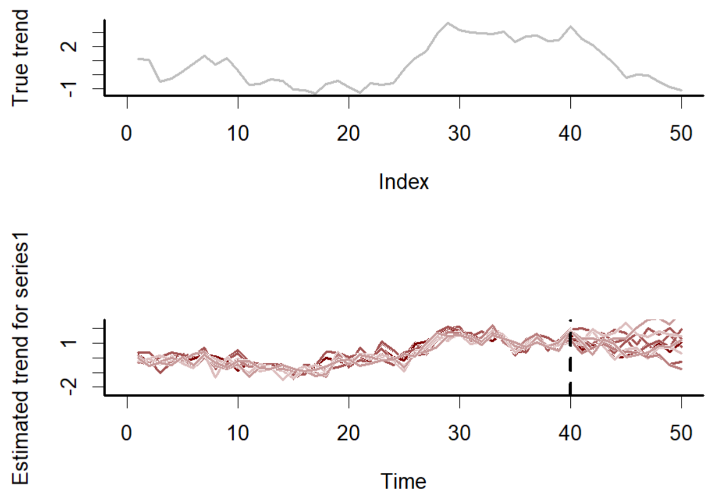
We can get a better idea of what is happening for the autoregressive observation model by plotting posterior realisations on the log scale. Here each line is a realisation, and we show these lines as a spaghetti plot to make it easier to visualise the diversity of possible forecast paths that are compatible with our model configuration.

```
plot(1, type = "n", bty = 'L',
     xlab = 'Time',
     ylab = 'GAMAR simulations (log scale)',
     xlim = c(0, NCOL(gam_sims)),
     ylim = range(log(gam_sims + 0.01)))
for(i in 1:30){
  lines(x = 1:NCOL(gam_sims),
        y = log(gam_sims[i,] + 0.01),
        col = 'white',
        lwd = 3)
  lines(x = 1:NCOL(gam_sims),
        y = log(gam_sims[i,] + 0.01),
        col = sample(c("#DCBCBC80",
                       "#C7999980",
                       "#B97C7C80",
                       "#A2505080",
                       "#7C000080"), 1),
        lwd = 2.75)
}
box(bty = 'L', lwd = 2)
```



`mvgam` offers options to plot realisations for most types of plots as well. Repeating the trend plot above (which automatically shows the expected latent trend on the log scale) shows that the dynamic GAM does not suffer from the same explosion towards infinity that the autoregressive model suffers from

```r
layout(matrix(1:2, ncol = 1, nrow = 2))
plot(trend[4:53], xlim = c(0, 50), type = 'l', lwd = 2, col = 'grey',
     ylab = 'True trend', bty = 'L')
box(bty = 'L', lwd = 2)
plot(m_mvgam, 'trend', newdata = data_test, realisations = TRUE,
     n_realisations = 10)
```



```r
layout(1)
```

From the plots above it is apparent that although some of the realisations are sensible, as soon as a path climbs upward it tends to skyrocket up towards infinity. It is sensible then to conclude that an autocorrelated observation model will be extremely sensitive to the measurement process (particularly when there is overdispersion, outliers or missing observations), which is not a problem for the latent temporal process model. With this information in mind, could we improve the autocorrelated observation model by using logs of lagged observations, rather than raw lagged observations?

```r
data.frame(y = observations) %>%
  dplyr::mutate(lag1 = log(lag(y) + 0.01),
                lag2 = log(lag(y, 2) + 0.01),
                lag3 = log(lag(y, 3) + 0.01)) %>%
  dplyr::mutate(cov1 = rnorm(53)) %>%
  dplyr::slice_tail(n = 50) -> sim_dat
sim_dat$time <- 1:50
data_train <- sim_dat[1:40,]
data_test <- sim_dat[41:50,]
m_mgcv <- gam(y ~ lag1 + lag2 + lag3,
              data = data_train,
              method = "REML", family = nb())
```

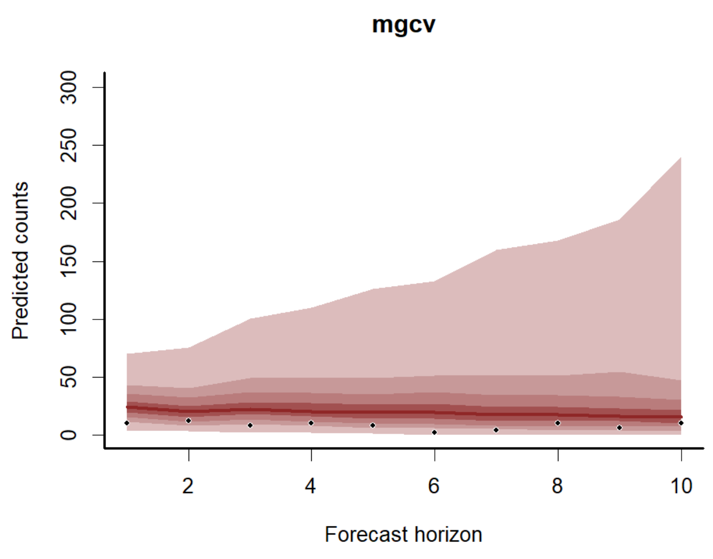
Compute the forecasts but use logged lags instead of raw lags

```r
coef_sim <- MASS::mvrnorm(1000, mu = m_mgcv$coefficients, Sigma = m_mgcv$Vp)
gam_sims <- matrix(NA, nrow = 1000, ncol = 10)
for(i in 1:1000){
  gam_sims[i,] <- recurse_ar3(m_mgcv,
                              coef_sim,
                              lagged_vals = c(data_test$lag1[1],
                                              data_test$lag2[1],
                                              data_test$lag3[1]),
                              log_lags = TRUE,
                              h = 10)
}
```

Evaluate the log-AR model against the dynamic GAM

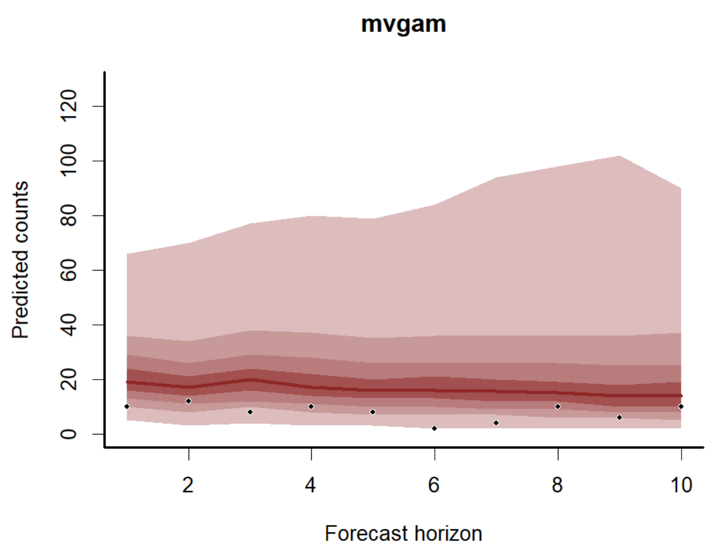
```r
plot_fc_horizon(gam_sims = gam_sims[,1:10], main = 'mgcv')
```

```
## Out of sample DRPS:

## [1] 91.19192

##
```



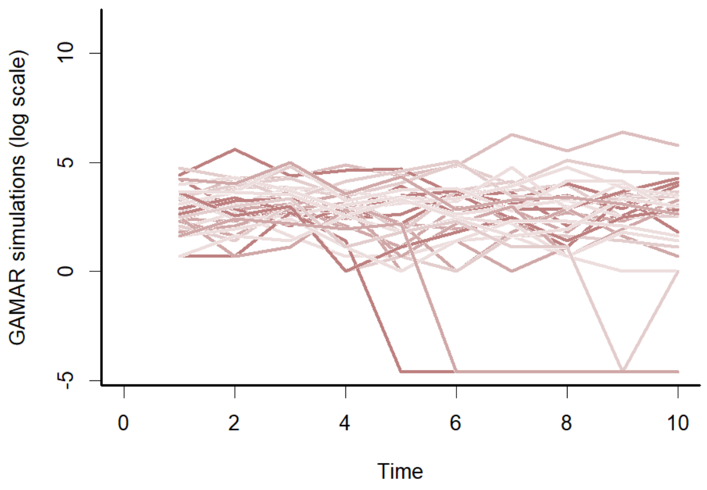
```r
plot_fc_horizon(gam_sims = MCMCvis::MCMCchains(m_mvgam$model_output,
                                                'ypred')[,(NROW(m_mvgam$obs_data)+1):
                                                            (NROW(m_mvgam$obs_data)+10)],
                main = 'mvgam')
```

```
## Out of sample DRPS:
## [1] 66.07028
##
```



We should also re-check the autoregressive model's realisations, which look much better now

```r
plot(1, type = "n", bty = 'L',
     xlab = 'Time',
     ylab = 'GAMAR simulations (log scale)',
     xlim = c(0, NCOL(gam_sims)),
     ylim = range(log(gam_sims + 0.01)))
for(i in 1:30){
  lines(x = 1:NCOL(gam_sims),
        y = log(gam_sims[i,] + 0.01),
        col = 'white',
        lwd = 3)
  lines(x = 1:NCOL(gam_sims),
        y = log(gam_sims[i,] + 0.01),
        col = sample(c("#DCBCBC80",
                       "#C7999980",
                       "#B97C7C80",
                       "#A2505080",
                       "#7C000080"), 1),
        lwd = 2.75)
}
box(bty = 'L', lwd = 2)
```



This version (with logged lags) performs much better than the raw lag version, with sensible forecasts that are not blowing up to infinity. In fact, this forecast is now more comparable to the dynamic GAM where the latent temporal process is estimated separately from the observation process, at least for this particular simulation. There are of course many other advantages to the state-space representation that is used by `mvgam`, including:

1. It is far easier to handle missing values (which will result in loss of many observations when fitting `mgcv` models, especially at higher order lags, due to missing outcomes AND missing predictors) and measurement error
2. Latent trend processes provide recursive expressions for h-step ahead prediction and historical filtering
3. Irregularly sampled observational time series can be readily modeled as the latent temporal process can simply be 'forecasted' over the missing observations
4. Multiple observation processes can depend on shared latent states

Fisheries landings example

Now for an empirical example of DGAMs, using historical data on US fisheries landings made available by a recent paper published in `Methods in Ecology and Evolution` (Smoothed dynamic factor analysis for identifying trends in multivariate time series; Ward et al 2021). The data consists of adjusted landings for 13 species or groups reported annually from multiple fisheries over a 39-year period (1981–2019)

```
data <- read.csv('https://raw.githubusercontent.com/fate-ewi/gpdfa/main/data/port_landings_table2.csv')
head(data)
```

First some data wrangling to clean the column names, create logged lags for the species of interest (the Ling cod) and gather values for an additional species that will be used as a (potentially nonlinear) covariate (the whiting). These two species were chosen in an ad-hoc fashion based on results of the above paper, which found that they demonstrated somewhat opposing patterns in their historical trends. Note that this is not meant to be a full in-depth analysis of these data, but rather a simple example to demonstrate how `mvgam` can be used for empirical series.

```
data %>%
  janitor::clean_names() %>%
  dplyr::mutate(y = lingcod) %>%
  dplyr::mutate(lag1 = log(lag(y) + 0.01),
                lag2 = log(lag(y, 2) + 0.01),
                lag3 = log(lag(y, 3) + 0.01)) %>%
  dplyr::select(y,
                lag1,
                lag2,
                lag3,
                p_whiting) %>%
  # The nonlinear covariate will be a smooth of standardised whiting landings
  dplyr::mutate(p_whiting = as.vector(scale(p_whiting))) %>%
  dplyr::slice_tail(n = 36) -> model_dat
model_dat$time <- 1:NROW(model_dat)
```

View the modelling dataframe and check for any `NAs` (which will be omitted by `mgcv` automatically)

```
head(model_dat)
```

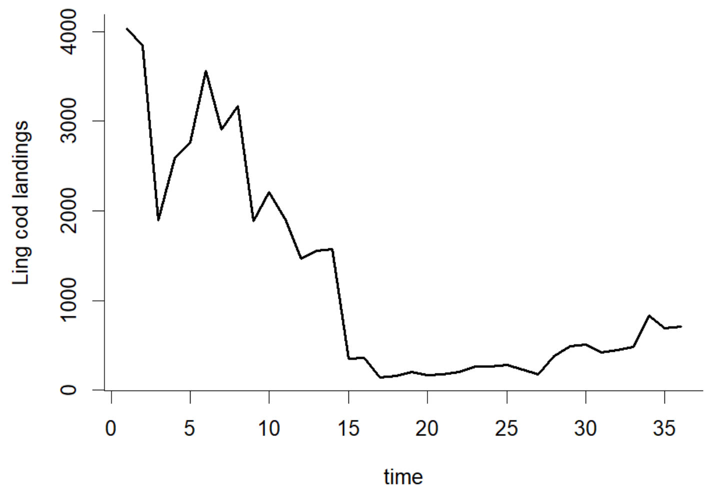
```
##      y     lag1     lag2     lag3 p_whiting time
## 1 4029 8.355147 8.253230 8.102589 -1.378056    1
## 2 3839 8.301276 8.355147 8.253230 -1.351442    2
## 3 1891 8.252970 8.301276 8.355147 -1.361176    3
## 4 2587 7.544866 8.252970 8.301276 -1.331000    4
## 5 2767 7.858258 7.544866 8.252970 -1.283989    5
## 6 3563 7.925523 7.858258 7.544866 -1.271578    6
```

```
anyNA(model_dat)
```

```
## [1] FALSE
```

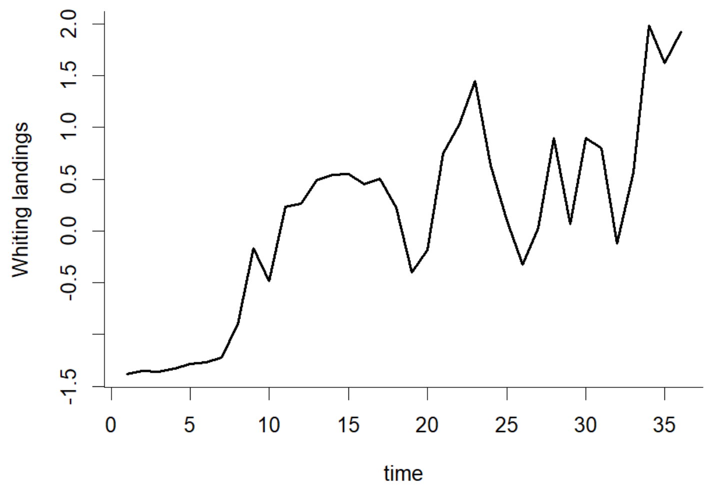
Plot the time series for the species of interest (the ling cod)

```
with(model_dat, plot(time, y, 'l', bty = 'L', lwd = 2, ylab = 'Ling cod landings'))
```



And for the covariate (whiting landings)

```
with(model_dat, plot(time, p_whiting, 'l', bty = 'L', lwd = 2, ylab = 'Whiting landings'))
```



Split into 75% train and 25% test periods

```
data_train <- model_dat[1:27,]
data_test <- model_dat[28:36,]
```

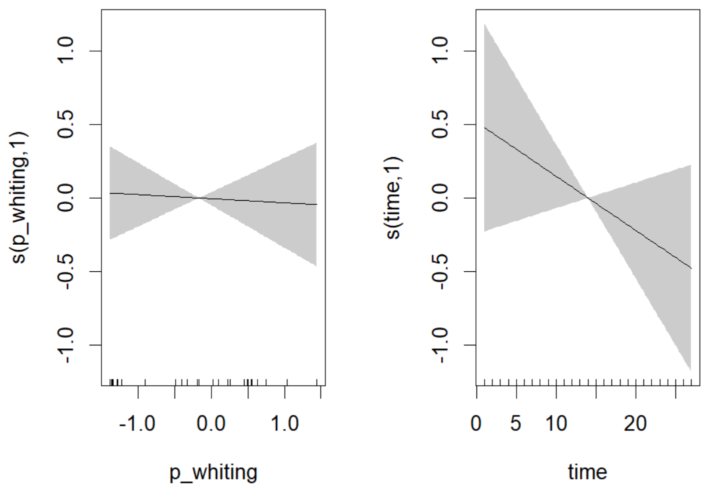
Fit an mgcv autocorrelated observation model, with an additional TPRS smooth function of time and a TRPS smooth function of whiting landings

```
m_mgcv <- gam(y ~ s(p_whiting, k = 5) +
                s(time, k = 12) +
                lag1 + lag2 + lag3,
              data = data_train,
              method = "REML", family = nb())
```

```r
summary(m_mgcv)
```

```
##
## Family: Negative Binomial(8.18)
## Link function: log
##
## Formula:
## y ~ s(p_whiting, k = 5) + s(time, k = 12) + lag1 + lag2 + lag3
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   1.8207     1.0265   1.774   0.0761 .
## lag1          0.7439     0.1870   3.978 6.96e-05 ***
## lag2          0.2532     0.2264   1.118   0.2634
## lag3         -0.2731     0.1912  -1.429   0.1530
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##               edf Ref.df Chi.sq p-value
## s(p_whiting)    1      1  0.046   0.831
## s(time)         1      1  1.842   0.175
##
## R-sq.(adj) =  0.817   Deviance explained = 92.2%
## -REML = 194.17  Scale est. = 1         n = 27
```

```r
plot(m_mgcv, pages=1, shade = TRUE)
```



The AR1 term is estimated as important, while the smooth functions of `time` and of `whiting landings` are regularized to flat lines. But how does the model forecast? Draw posterior coefficients and generate the iterative `mgcv` forecast

```r
coef_sim <- gam.mh(m_mgcv)$bs

# Forecast 9 years ahead
```

```
gam_sims <- matrix(NA, nrow = 1000, ncol = 9)
for(i in 1:1000){
  gam_sims[i,] <- recurse_ar3(m_mgcv,
                              coef_sim,
                              lagged_vals = c(data_test$lag1[1],
                                              data_test$lag2[1],
                                              data_test$lag3[1]),
                              log_lags = TRUE,
                              h = 9)
}
```

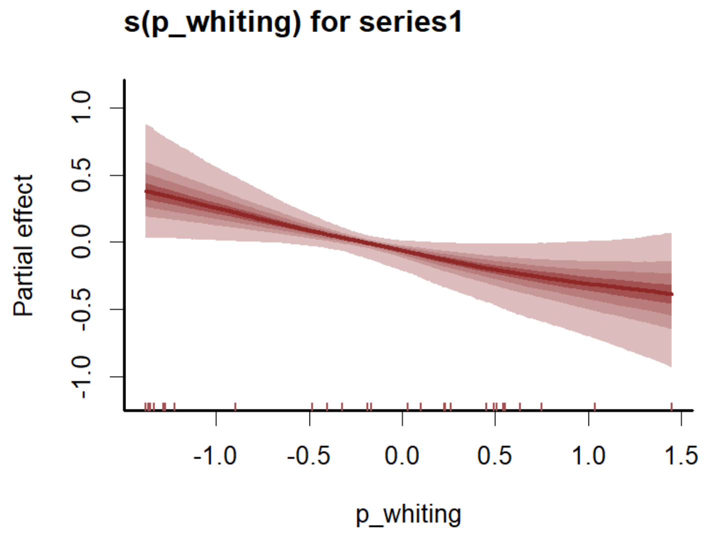
Now fit an `mvgam` model where the latent trend is modelled as an `AR3` process. We will use `Stan` for this example, which generally explores the joint posterior much more efficiently (and more fully) than `JAGS`

```
m_mvgam <- mvgam(y ~ s(p_whiting, k = 5),
                 data = data_train,
                 newdata = data_test,
                 family = 'nb',
                 trend_model = 'AR3',
                 use_stan = TRUE,
                 chains = 4,
                 burnin = 500)
```

```
## Running MCMC with 4 parallel chains...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 finished in 3.1 seconds.
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 3.4 seconds.
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 finished in 3.6 seconds.
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 4.4 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 3.6 seconds.
## Total execution time: 4.6 seconds.
```

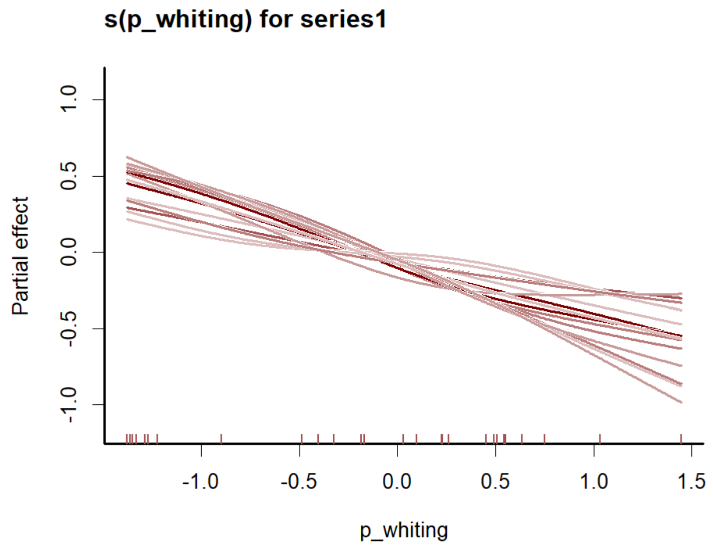
Inference on `mvgam` smooth functions can be conducted using posterior predictions. This model clearly finds a negative relationship between whiting landings and ling cod landings

```
plot(m_mvgam, 'smooths')
```



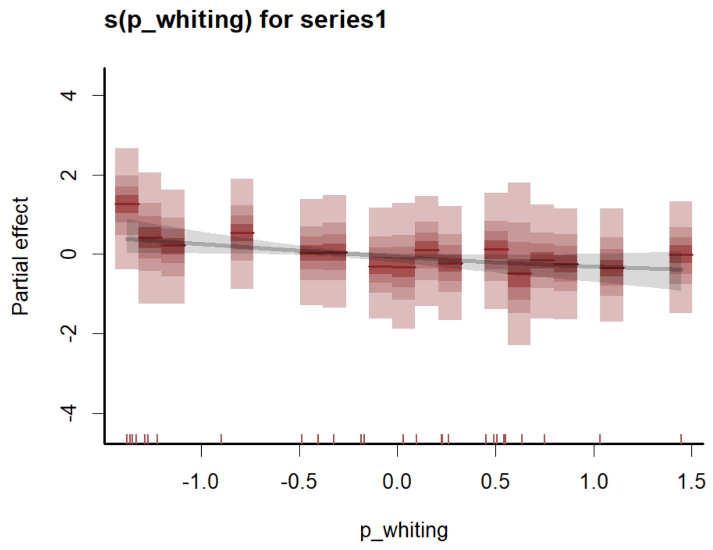
Inspect posterior realisations of the smooth function

```
plot(m_mvgam, 'smooths', realisations = TRUE)
```



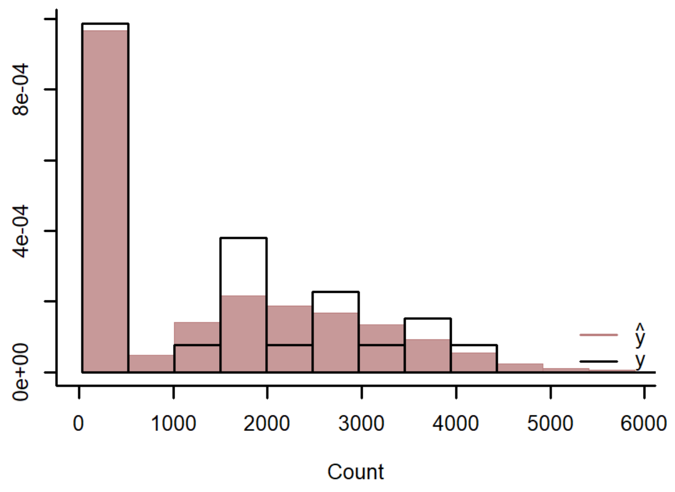
The `plot_mvgam_smooth` function allows more flexibility for plotting smooth functions, including an ability to supply `newdata` for plotting posterior marginal simulations. Overlay the partial Dunn-Smyth residuals on the smooth plot for `p_whiting`. These residuals suggest there is still structure left in the data and that perhaps the smooth for this term is not as flexible as it could be (i.e. we would be justified to try this model again by increasing `k` for the smooth term). But we will leave it as-is for now

```
plot_mvgam_smooth(m_mvgam, series = 1, smooth = 'p_whiting', residuals = TRUE)
```



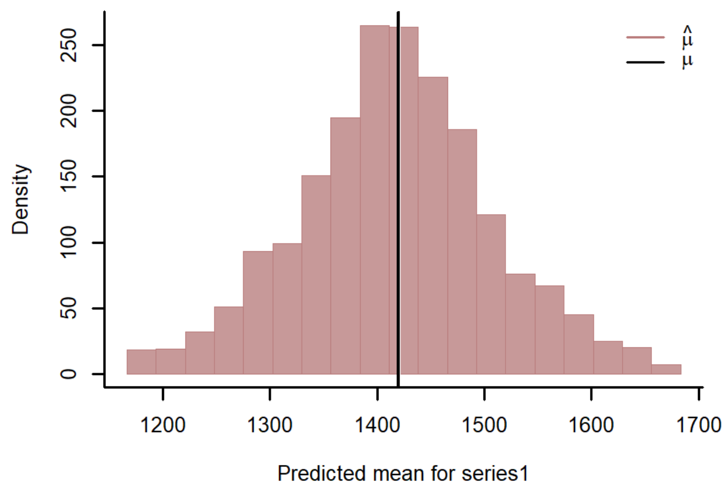
We can also perform a series of posterior predictive checks (using the `ppc()` function) to see if the model is able to simulate data for the training period that looks realistic and unbiased. First, examine simulated histograms for posterior predictions (`yhat`) and compare to the observations (`y`)

```
ppc(m_mvgam, series = 1, type = 'hist', legend_position = 'bottomright')
```



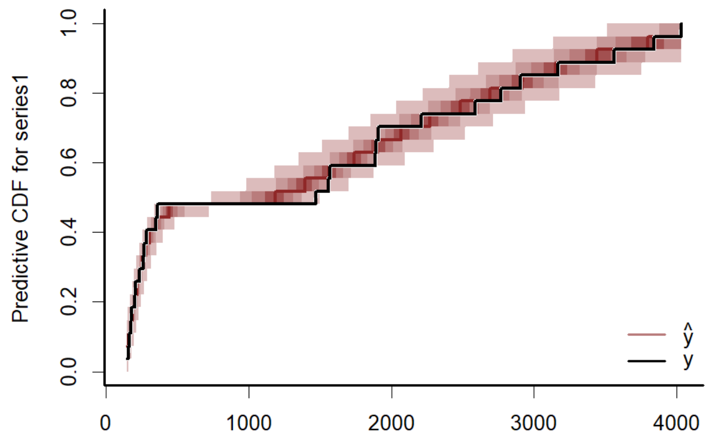
Now plot the distribution of predicted means compared to the observed mean

```r
ppc(m_mvgam, series = 1, type = 'mean')
```



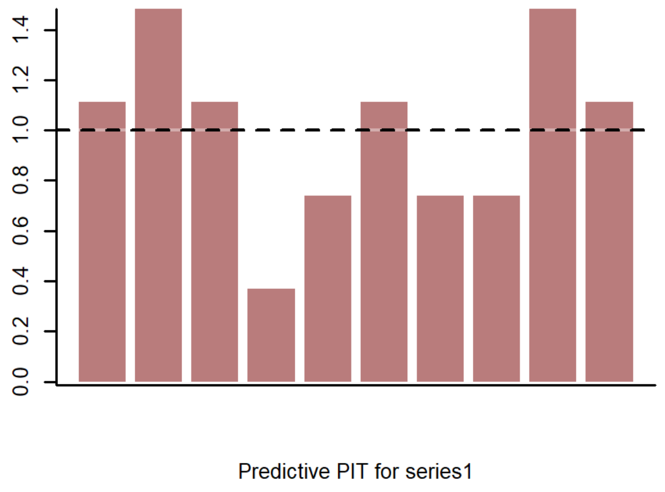
Next examine simulated empirical Cumulative Distribution Functions (CDF) for posterior predictions (`yhat`) and compare to the CDF of the observations (`y`)

```r
ppc(m_mvgam, series = 1, type = 'cdf')
```



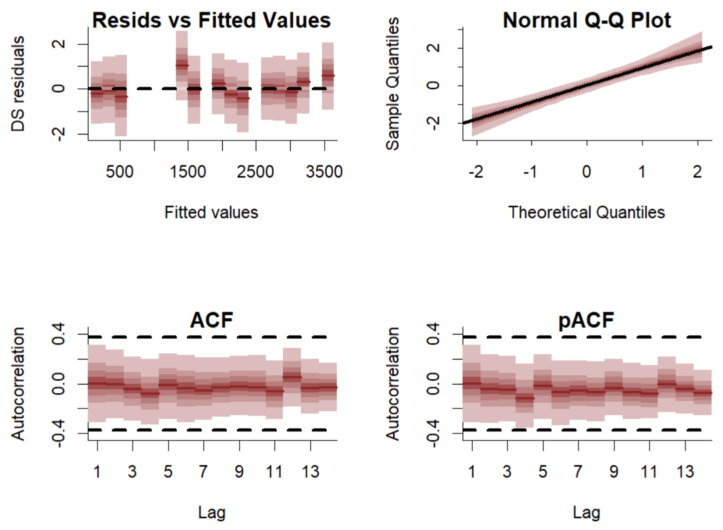
Finally look for any biases in predictions by examining a Probability Integral Transform (PIT) histogram. If our predictions are not biased one way or another (i.e. not consistently under- or over-predicting), this histogram should look roughly uniform

```
ppc(m_mvgam, series = 1, type = 'pit')
```



All of these plots indicate the model is well calibrated against the training data, with no apparent pathological behaviors exhibited. Now for some investigation of the estimated relationships and forecasts. We can also perform residual diagnostics using randomised quantile (Dunn-Smyth) residuals. These look reasonable overall, though there is some autocorrelation at recent lags left in the residuals for this series

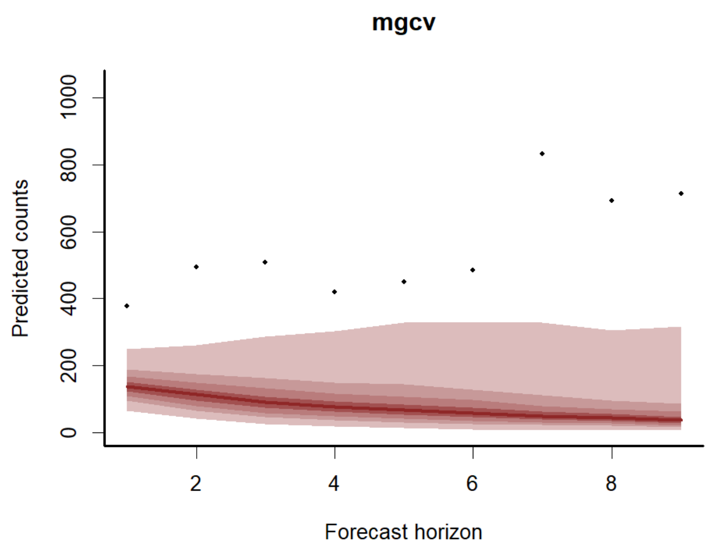
```
plot(m_mvgam, series = 1, type = 'residuals')
```



Ok so the model is doing well when fitting against the training data, but how are its forecasts? We can now evaluate the forecasts of the two competing models just as we did above for the simulated data

```r
plot_fc_horizon(gam_sims = gam_sims, main = 'mgcv')
```

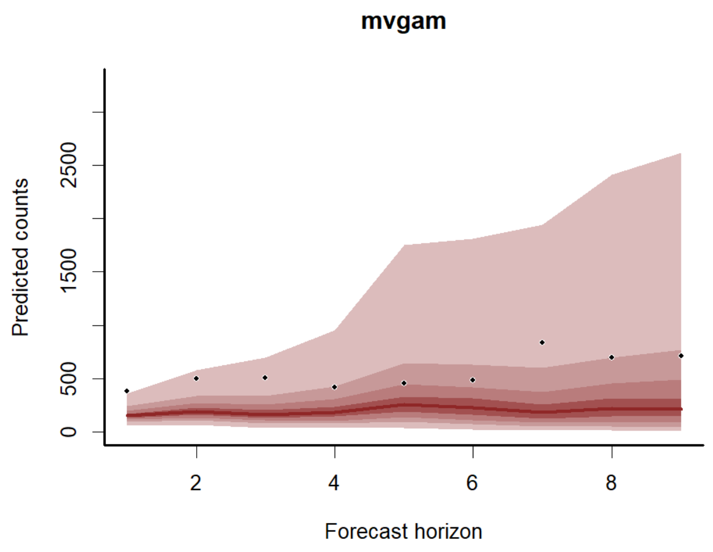
```
## Out of sample DRPS:

## [1] 3649.149

##
```



```r
plot_fc_horizon(gam_sims = MCMCvis::MCMCchains(m_mvgam$model_output,
                                                'ypred')[,(NROW(m_mvgam$obs_data)+1):
                                                         (NROW(m_mvgam$obs_data)+9)],
                main = 'mvgam')
```

```
## Out of sample DRPS:
```

```
## [1] 1892.626

##
```



Both models fail to anticipate the upward swing in landings following the end of the training data, but as with the simulations above, the autocorrelated observation model fitted by `mgcv` is providing inferior forecasts compared to the dynamic trend model estimated by `mvgam`. What if we try to improve the smooth function of time to do improve the forecasts? To try and capture the long-term trend we can use a `B` spline with multiple penalties, following the excellent example by Gavin Simpson about extrapolating with smooth terms. This is similar to what we might do when trying to forecast ahead from a more wiggly function, as `B` splines have useful properties by allowing the penalty to be extended into the range of values we wish to predict (in this case, the years in `data_test`).

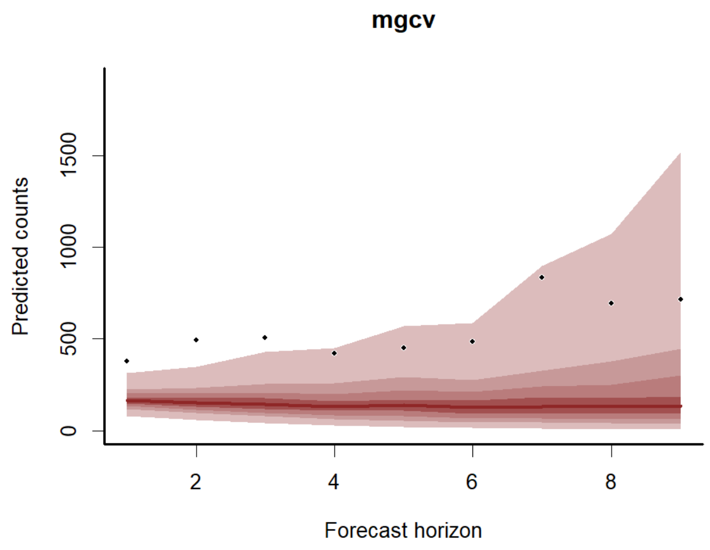
```
m_mgcv <- gam(y ~ s(p_whiting, k = 5) +
                s(time, bs = "bs", m = c(2, 1)) +
                lag1 + lag2 + lag3,
              knots = list(time = c(min(data_train$time) - 1,
                                    min(data_train$time),
                                    max(data_train$time),
                                    max(data_test$time))),
              data = data_train,
              method = "REML", family = nb())
```

Forecast and evaluate against the dynamic GAM

```
coef_sim <- gam.mh(m_mgcv)$bs
gam_sims <- matrix(NA, nrow = 1000, ncol = 9)
for(i in 1:1000){
  gam_sims[i,] <- recurse_ar3(m_mgcv,
                              coef_sim,
                              lagged_vals = c(data_test$lag1[1],
                                              data_test$lag2[1],
                                              data_test$lag3[1]),
                              log_lags = TRUE,
                              h = 9)
}
```

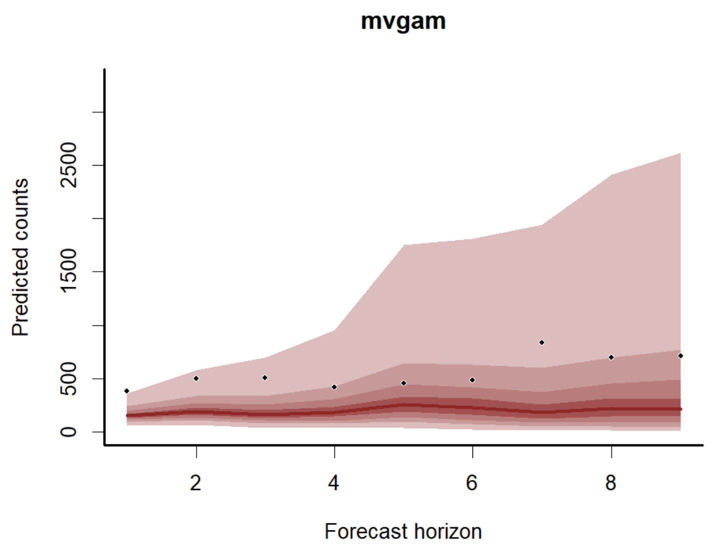
```
plot_fc_horizon(gam_sims = gam_sims, main = 'mgcv')
```

```
## Out of sample DRPS:
## [1] 2586.687
##
```



```
plot_fc_horizon(gam_sims = MCMCvis::MCMCchains(m_mvgam$model_output,
                                              'ypred')[,(NROW(m_mvgam$obs_data)+1):
                                                        (NROW(m_mvgam$obs_data)+9)],
                main = 'mvgam')
```

```
## Out of sample DRPS:
## [1] 1892.626
##
```



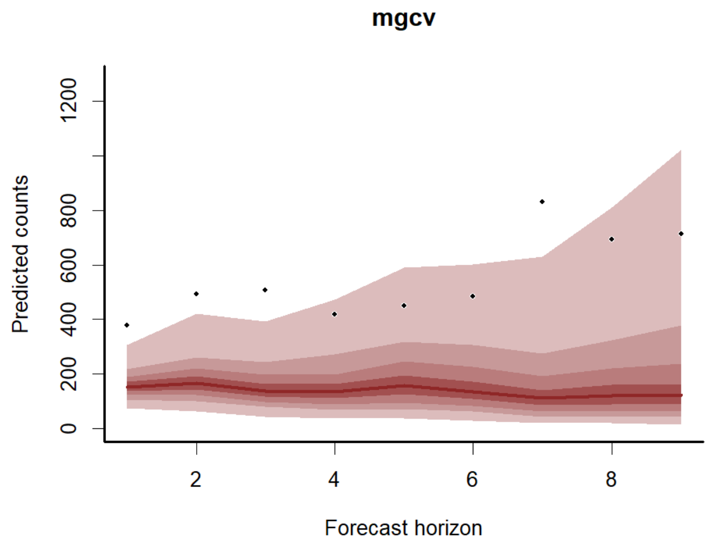
This model shows an improvement in forecasting ability as the uncertainty in the temporal smooth function is more realistically growing into the future. But again, the dynamic GAM is superior in forecasting ability. Perhaps the `AR` terms are doing more harm than good for the model's forecasts? For a final comparison, drop the `AR` functions

```r
m_mgcv <- gam(y ~ s(p_whiting, k = 5) +
                s(time, bs = "bs", m = c(2, 1), k = 15),
              knots = list(time = c(min(data_train$time) - 1,
                                    min(data_train$time),
                                    max(data_train$time),
                                    max(data_test$time))),
              data = data_train,
              method = "REML", family = nb())
```

```r
coef_sim <- gam.mh(m_mgcv)$bs
gam_sims <- matrix(NA, nrow = 1000, ncol = 9)
for(i in 1:1000){
  gam_sims[i,] <- recurse_ar3(m_mgcv,
                              coef_sim,
                              lagged_vals = c(data_test$lag1[1],
                                              data_test$lag2[1],
                                              data_test$lag3[1]),
                              log_lags = TRUE,
                              h = 9)
}
```

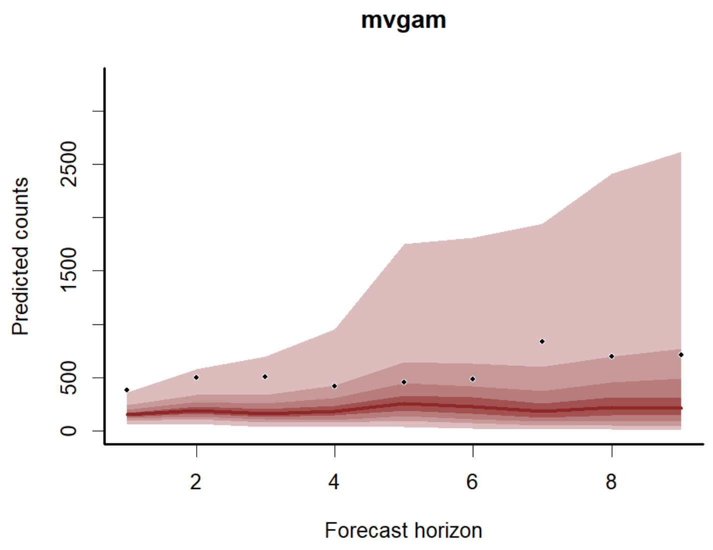
```r
plot_fc_horizon(gam_sims = gam_sims, main = 'mgcv')
```

```
## Out of sample DRPS:

## [1] 2651.224

##
```



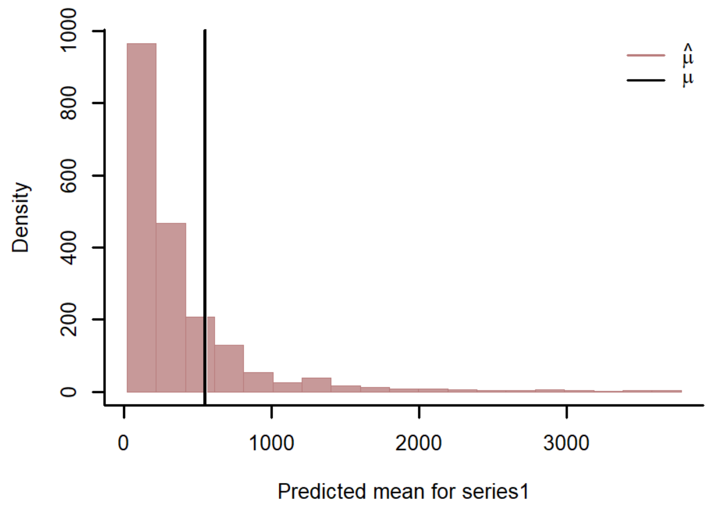
```r
plot_fc_horizon(gam_sims = MCMCvis::MCMCchains(m_mvgam$model_output,
                                               'ypred')[,(NROW(m_mvgam$obs_data)+1):
                                                          (NROW(m_mvgam$obs_data)+9)],
                    main = 'mvgam')
```

```
## Out of sample DRPS:
## [1] 1892.626
##
```

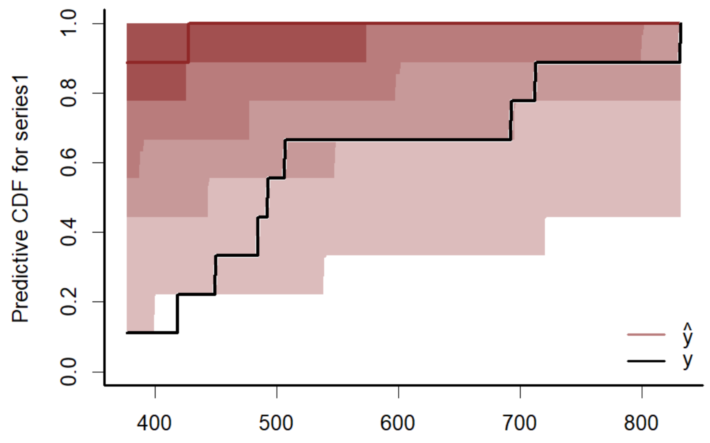


The dynamic GAM is difficult to beat for this example. But what other utilities does the `mvgam` package make available? We can also re-do the posterior predictive checks, but this time focusing only on the out of sample period. This will give us better insight into how the model is performing and whether it is able to simulate realistic and unbiased future values

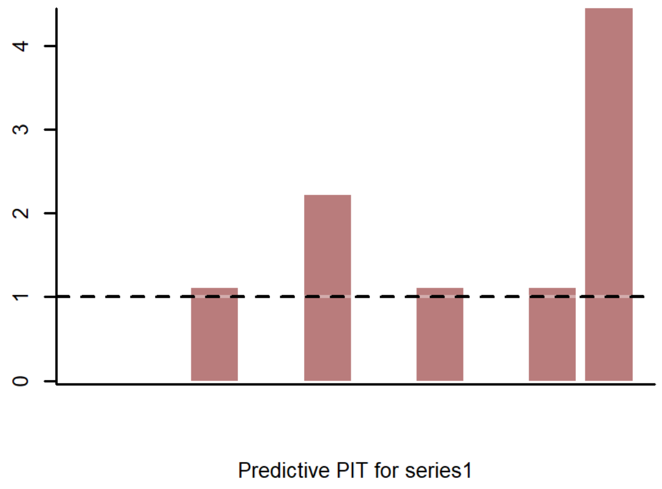
```
ppc(m_mvgam, series = 1, type = 'mean', newdata = data_test)
```



```
ppc(m_mvgam, series = 1, type = 'cdf', newdata = data_test)
```



```
ppc(m_mvgam, series = 1, type = 'pit', newdata = data_test)
```



There are some problems with the way this model is generating future predictions, so we would need to perform a more rigorous and principled model development strategy to improve our model's predictive abilities. But there are two other utilities available in `mvgam` that are worth highlighting. The first is model comparison. Benchmarking against "null" models is a very important part of evaluating a proposed forecast model. After all, if our complex dynamic model can't generate better predictions then a random walk or mean forecast, is it really telling us anything new about the data-generating process? Here we examine the model comparison utilities in `mvgam`. Here we illustrate how this can be done in `mvgam` by fitting a basic random walk model with a Negative Binomial observation process. No smooth terms are included here, the observation model simply has an intercept term.

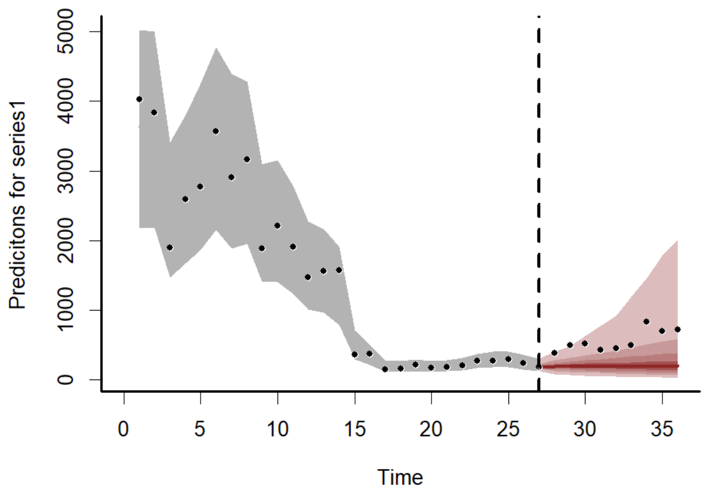
```
m_mvgam2 <- mvgam(y ~ 1,
                  data = data_train,
                  newdata = data_test,
                  family = 'nb',
                  trend_model = 'RW',
                  use_stan = TRUE,
                  chains = 4)
```

```
## Running MCMC with 4 parallel chains...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 finished in 0.9 seconds.
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 finished in 1.0 seconds.
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 finished in 1.1 seconds.
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 1.2 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 1.1 seconds.
## Total execution time: 1.4 seconds.
```

Look at this model's proposed forecast, which incidentally isn't too bad compared to the original `AR3` model

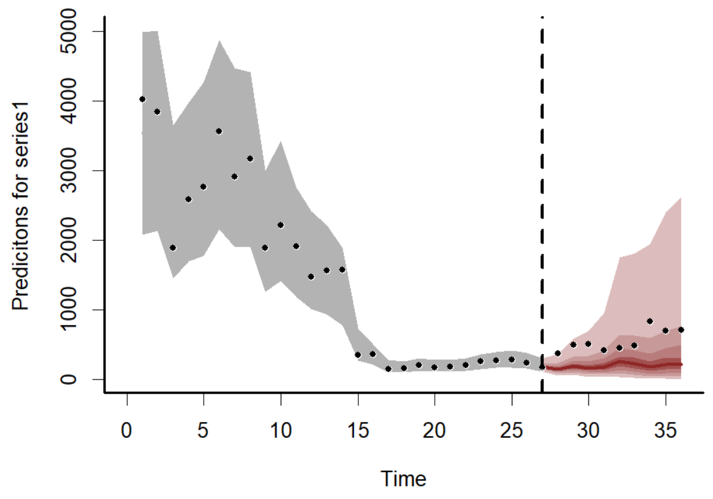
```r
plot(m_mvgam2, series = 1, type = 'forecast', newdata = data_test)
```

```
## Out of sample DRPS:

## [1] 2005.684

##
```



```r
plot(m_mvgam, series = 1, type = 'forecast', newdata = data_test)
```

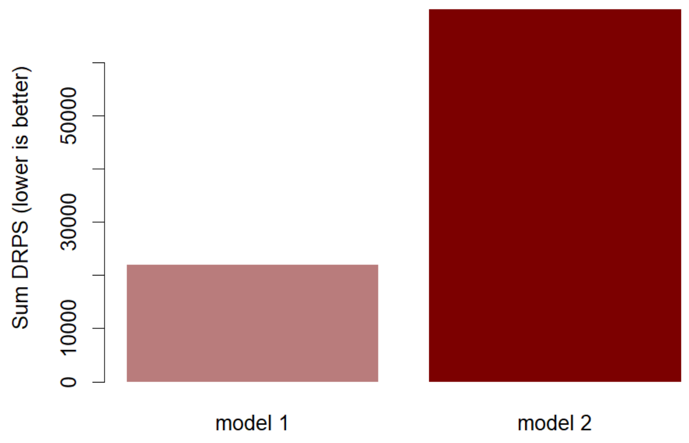
```
## Out of sample DRPS:

## [1] 1892.626

##
```

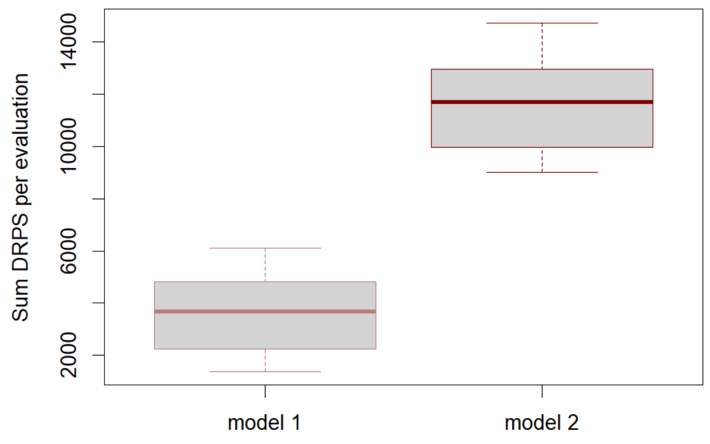


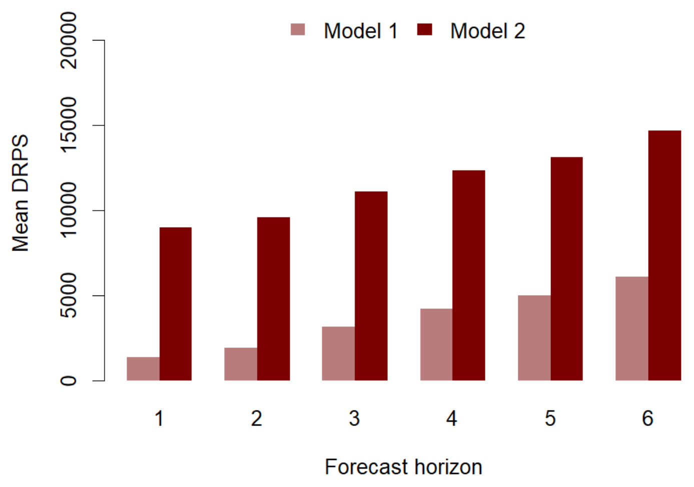
Now we will showcase how different dynamic models can be compared using rolling probabilistic forecast evaluation, which is especially useful if we don't already have out of sample observations for comparing forecasts. This function sets up a sequence of evaluation timepoints along a rolling window within the training data to evaluate 'out-of-sample' forecasts. The trends are rolled forward a total of `fc_horizon` timesteps according to their estimated state space dynamics to generate an 'out-of-sample' forecast that is evaluated against the true observations in the `horizon` window. We are therefore simulating a situation where the model's parameters had already been estimated but we have only observed data up to the evaluation timepoint and would like to generate forecasts that consider the possible future paths for the latent trends and the true observed values for any other covariates in the `horizon` window. Evaluation involves calculating the Discrete Rank Probability Score and a binary indicator for whether or not the true value lies within the forecast's 90% prediction interval. For this test we compare the two models on the exact same sequence of 30 evaluation points using `horizon = 6`

```
compare_mvgams(model1 = m_mvgam, model2 = m_mvgam2, fc_horizon = 6,
               n_evaluations = 30, n_cores = 3)
```

```
## DRPS summaries per model (lower is better)
##              Min.  1st Qu.    Median      Mean  3rd Qu.      Max.
## Model 1 1397.862 2251.666  3720.807  3650.961  4833.97  6099.316
## Model 2 8991.095 9967.357 11718.340 11643.858 12941.18 14702.883
##
## 90% interval coverages per model (closer to 0.9 is better)
## Model 1 0.8833333
## Model 2 0.4722222
```


Sum DRPS (lower is better)
0
10000
30000
50000
model 1
model 2


Sum DRPS per evaluation
2000
6000
10000
14000
model 1
model 2



The series of plots generated by `compare_mvgams` clearly show that the first dynamic model generates better predictions. In each plot, DRPS for the forecast `horizon` is lower for the first model than for the second model. This kind of evaluation is often more appropriate for forecast models than complexity-penalising fit metrics such as `AIC` or `BIC` Now we proceed by exploring how forecast distributions from an `mvgam` object can be automatically updated in light of new incoming observations. This works by generating a set of "particles" that each captures a unique proposal about the current state of the system (in this case, the current estimate of the latent trend component). The next observation in `data_assim` is assimilated and particles are weighted by how well their proposal (i.e. their proposed forecast, prior to seeing the new data) matched the new observations. For univariate models such as the ones we've fitted so far, this weight is represented by the proposal's Negative Binomial log-likelihood. For multivariate models, a multivariate composite likelihood is used for weights. Once weights are calculated, we use importance sampling to update the model's forecast distribution for the remaining forecast horizon. Begin by initiating a set of `20000` particles by assimilating the next observation in `data_test` and storing the particles in the default location (in a directory called `particles` within the working directory)

```
pfilter_mvgam_init(object = m_mvgam, n_particles = 20000,
                   n_cores = 3, data_assim = model_dat[28,])
```

```
## Saving particles to pfilter/particles.rda
##  ESS = 2209.862
```

Now we are ready to run the particle filter. This function will assimilate the next two out of sample observations in `data_test` and update the forecast after each assimilation step. This works in an iterative fashion by calculating each particle's weight, then using a kernel smoothing algorithm to "pull" low weight particles toward the high-likelihood space before assimilating the next observation. The strength of the kernel smoother is controlled by `kernel_lambda`, which in our experience works well when left to the default of `1`. If the Effective Sample Size of particles drops too low, suggesting we are putting most of our belief in a very small set of particles, an automatic resampling step is triggered to increase particle diversity and reduce the chance that our forecast intervals become too narrow and incapable of adapting to changing conditions

```
pfilter_mvgam_online(data_assim = model_dat[29:30, ], n_cores = 3,
                     kernel_lambda = 1, use_resampling = TRUE)
```

```
## Assimilating the next 2 observations
##
## Effective sample size is 2592.296 ...
```

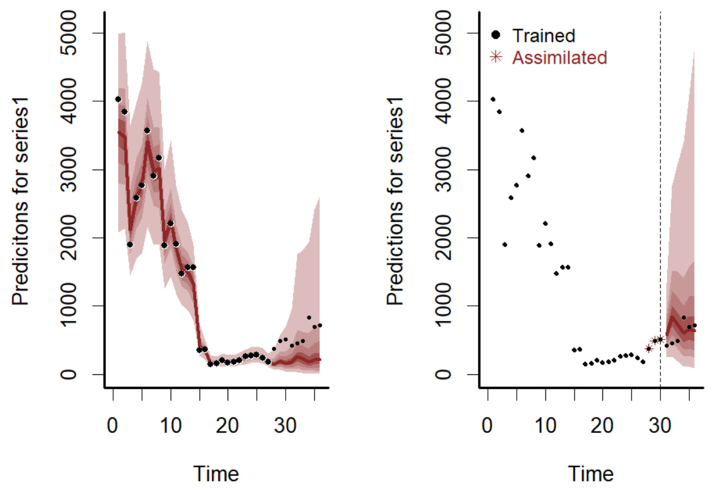
```
##
## Resampling particles ...
## Smoothing particles ...
##
## Effective sample size is 7906.202 ...
##
## Resampling particles ...
## Smoothing particles ...
##
## Last assimilation time was 30
##
## Saving particles to pfilter/particles.rda
##  ESS = 20000
```

Once assimilation is complete, generate the updated forecast from the particles using the covariate information in remaining `data_test` observations. This function is designed to hopefully make it simpler to assimilate observations, as all that needs to be provided once the particles are initiated as a dataframe of test data in exactly the same format as the data that were used to train the initial model. If no new observations are found (observations are arranged by `time` so the consistent indexing of these two variables is very important!) then the function returns a `NULL` and the particles remain where they are in state space.

```
pfilter_fc <- pfilter_mvgam_fc(file_path = "pfilter",
                               n_cores = 3, newdata = model_dat[31:36,],
                               return_forecasts = TRUE,
                               ylim = c(0, 5100))
```

Compare the updated forecast to the original forecast to see how it has changed in light of the most recent observations. As with the `plot_mvgam_smooth()` function, the `plot_mvgam_fc()` function has more flexibility for plotting posterior predictive distributions than the generic `plot.mvgam()` S3 function

```
layout(matrix(1:2,nrow=1,ncol=2))
plot_mvgam_fc(m_mvgam,
     ylim = c(0, 5100))
points(model_dat$y, pch = 16, cex = 0.4)
pfilter_fc$fc_plots$series1()
points(model_dat$y, pch = 16, cex = 0.4)
```



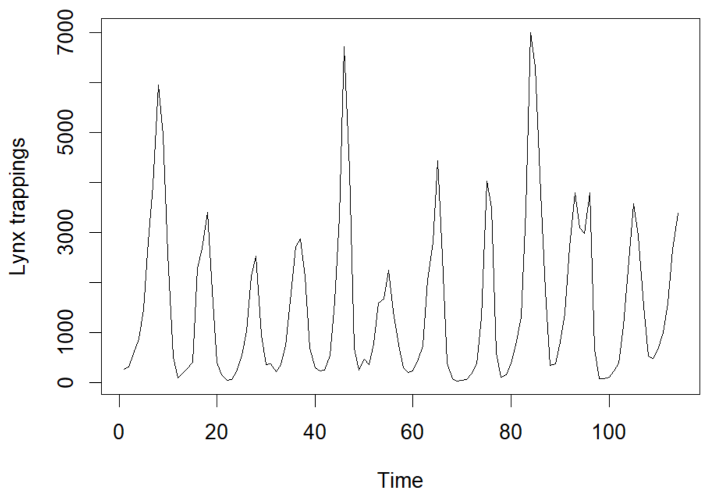
Here it is apparent that the distribution has shifted upward in light of the 3 observations that have been assimilated. This is an advantageous way of allowing a model to slowly adapt to new conditions while breaking free of restrictive assumptions about residual distributions. See some of the many particle filtering lectures by Nathaniel Osgood for more details. Remove the particles from their stored directory when finished

```
unlink('pfilter', recursive = T)
```

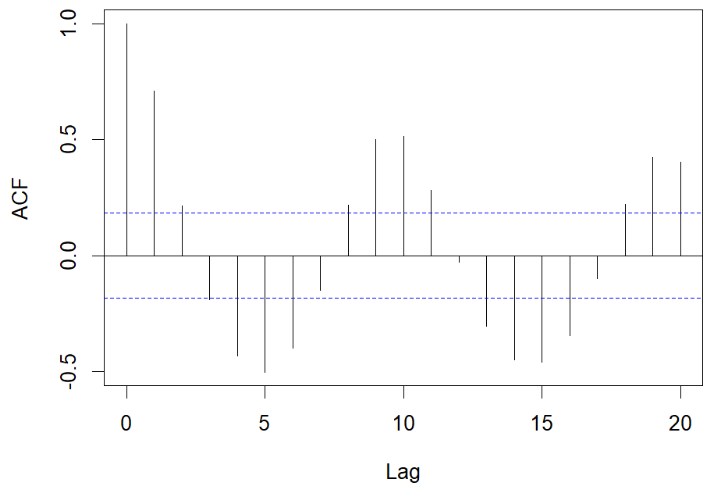
Lynx example

For our next univariate example, we will again pursue how challenging it can be to forecast ahead with conventional GAMs and how `mvgam` overcomes these challenges. We begin by replicating the lynx analysis from 2018 Ecological Society of America workshop on GAMs that was hosted by Eric Pedersen, David L. Miller, Gavin Simpson, and Noam Ross, with some minor adjustments. First, load the data and plot the series as well as its estimated autocorrelation function

```
data(lynx)
lynx_full = data.frame(year = 1821:1934,
                       population = as.numeric(lynx),
                       time = 1:NROW(lynx))
plot(lynx_full$population, type = 'l', ylab = 'Lynx trappings',
     xlab = 'Time')
```

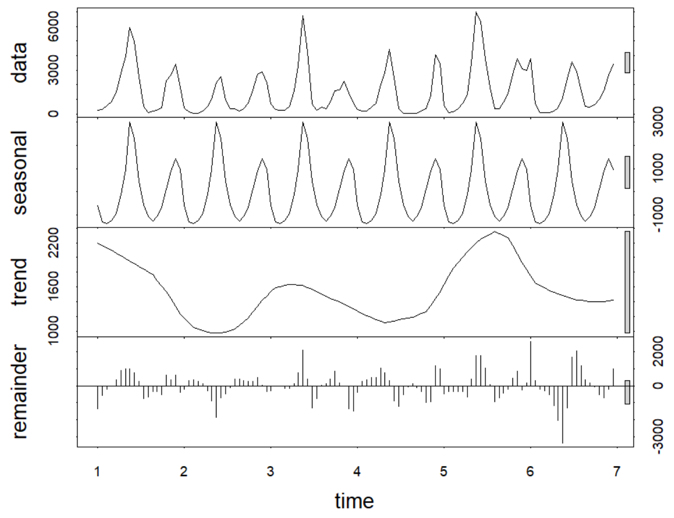


```
acf(lynx_full$population, main = '')
```



There is a clear ~19-year cyclic pattern to the data, so I create a `season` term that can be used to model this effect and give a better representation of the data generating process. I also create a new `year` term that represents which long-term cycle each observation is in

```
plot(stl(ts(lynx_full$population, frequency = 19), s.window = 'periodic'))
```



```
lynx_full$season <- (lynx_full$year %%19) + 1
cycle_ends <- c(which(lynx_full$season == 19), NROW(lynx_full))
cycle_starts <- c(1, cycle_ends[1:length(which(lynx_full$season == 19))] + 1)
cycle <- vector(length = NROW(lynx_full))
for(i in 1:length(cycle_starts)){
  cycle[cycle_starts[i]:cycle_ends[i]] <- i
}
lynx_full$year <- cycle
```

Add lag indicators needed to fit the nonlinear lag models that gave the best one step ahead point forecasts in the ESA workshop example. As in the example, we specify the `default` argument in the `lag` function as the mean log population.

```
mean_pop_l = mean(log(lynx_full$population))
lynx_full = dplyr::mutate(lynx_full,
                   popl = log(population),
                   lag1 = dplyr::lag(popl,1, default = mean_pop_l),
                   lag2 = dplyr::lag(popl,2, default = mean_pop_l),
                   lag3 = dplyr::lag(popl,3, default = mean_pop_l),
                   lag4 = dplyr::lag(popl,4, default = mean_pop_l),
                   lag5 = dplyr::lag(popl,5, default = mean_pop_l),
                   lag6 = dplyr::lag(popl,6, default = mean_pop_l))
```

For `mvgam` models, the response needs to be labelled `y` and we also need an indicator of the series name as a `factor` variable

```
lynx_full$y <- lynx_full$population
lynx_full$series <- factor('series1')
```

Split the data into training (first 40 years) and testing (next 10 years of data) to evaluate multi-step ahead forecasts

```
lynx_train = lynx_full[1:40, ]
lynx_test = lynx_full[41:50, ]
```

The best-forecasting model in the course was with nonlinear smooths of lags 1 and 2; we use those here is that we also include a cyclic smooth for the 19-year cycles as this seems like an important feature, as well as a yearly smooth for the long-term trend. Following the information about spline extrapolation above, we again fit a cubic `B` spline for the trend with a mix of penalties to try and reign in wacky extrapolation behaviours, and we extend the penalty to cover the years that we wish to predict. This will hopefully give us better uncertainty estimates for the forecast. In this example we assume the observations are Poisson distributed

```
lynx_mgcv = gam(population ~
                  s(season, bs = 'cc', k = 19) +
                  s(year, bs = 'bs', m = c(3, 2, 1, 0)) +
                  s(lag1, k = 5) +
                  s(lag2, k = 5),
               knots = list(season = c(0.5, 19.5),
                            year = c(min(lynx_train$year - 1),
                                     min(lynx_train$year),
                                     max(lynx_test$year),
                                     max(lynx_test$year) + 1)),
                data = lynx_train, family = "poisson",
                method = "REML")
```

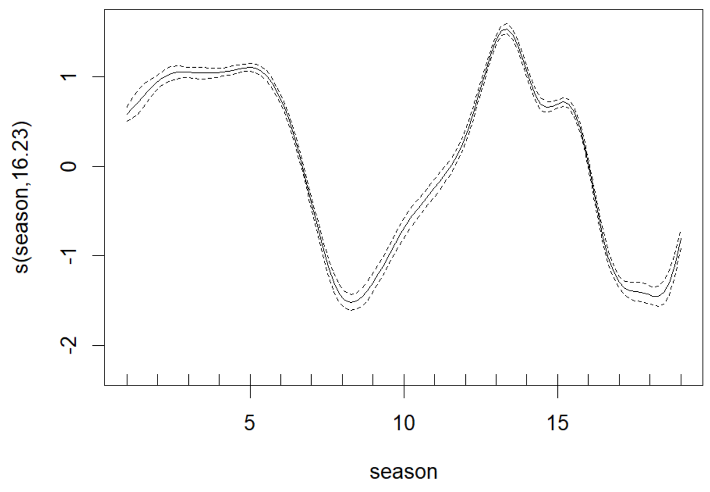
Inspect the model's summary and estimated smooth functions for the season, year and lag terms

```
summary(lynx_mgcv)
```

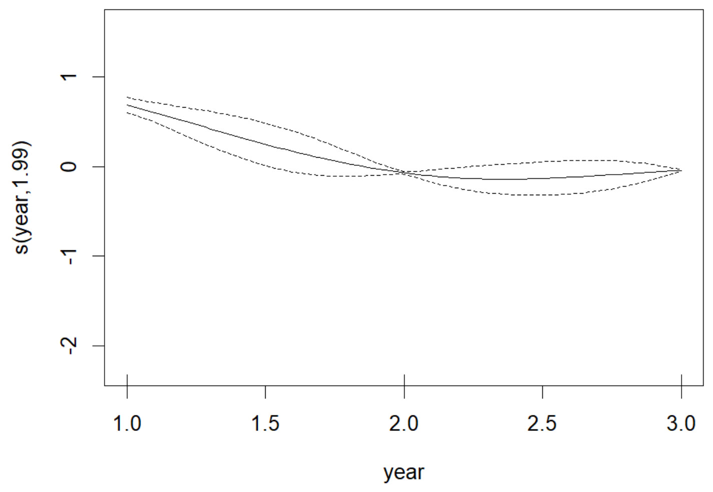
```
##
## Family: poisson
## Link function: log
##
## Formula:
## population ~ s(season, bs = "cc", k = 19) + s(year, bs = "bs",
##     m = c(3, 2, 1, 0)) + s(lag1, k = 5) + s(lag2, k = 5)
##
## Parametric coefficients:
```

```
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 6.666631   0.007511   887.6   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##              edf Ref.df Chi.sq p-value
## s(season) 16.229 17.000 6770.2  <2e-16 ***
## s(year)    1.990  2.000  244.2  <2e-16 ***
## s(lag1)    3.984  3.999  712.0  <2e-16 ***
## s(lag2)    3.892  3.993  488.1  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.967   Deviance explained = 97.7%
## -REML = 880.28  Scale est. = 1         n = 40
```

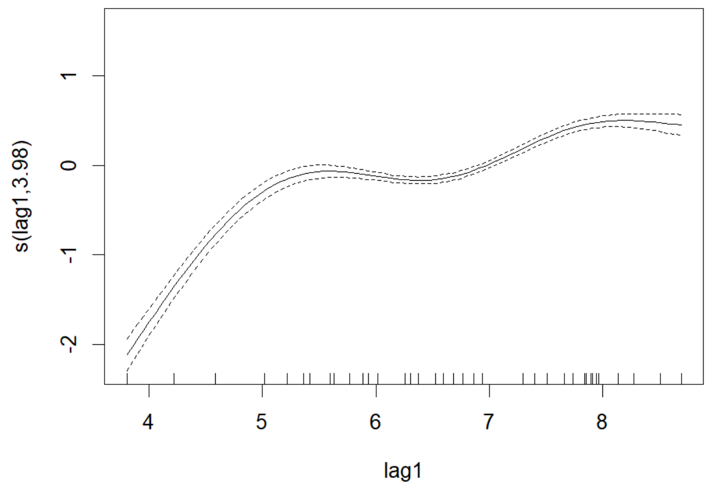
```
plot(lynx_mgcv, select = 1)
```



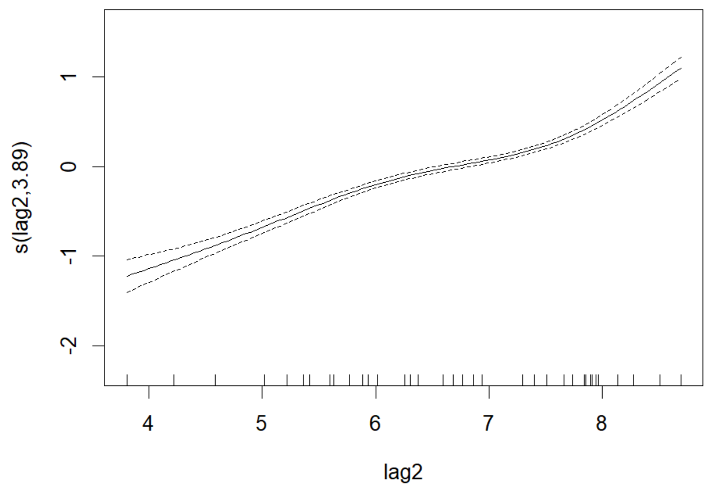
```
plot(lynx_mgcv, select = 2)
```



```r
plot(lynx_mgcv, select = 3)
```



```r
plot(lynx_mgcv, select = 4)
```



This model captures most of the deviance in the series and the functions are all confidently estimated to be non-zero and non-flat. So far, so good. Now for some forecasts for the out of sample period. First we must take posterior draws of smooth beta coefficients to incorporate the uncertainties around smooth functions when simulating forecast paths

```
coef_sim <- gam.mh(lynx_mgcv)$bs
```

Now we define a function to perform forecast simulations from the nonlinear lag model in a recursive fashion. Using starting values for the last two lags, the function will iteratively project the path ahead with a random sample from the model's coefficient posterior

Click for definition of `recurse_nonlin`

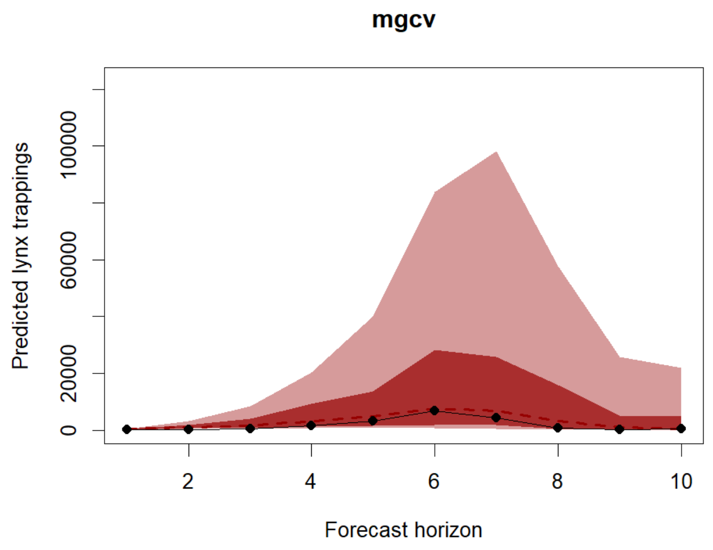
```
recurse_nonlin = function(model, coef_sim, lagged_vals, h){
  # Initiate state vector
  states <- rep(NA, length = h + 2)
  # Last two values of the conditional expectations begin the state vector
  states[1] <- as.numeric(exp(lagged_vals[2]))
  states[2] <- as.numeric(exp(lagged_vals[1]))
  # Get a random sample of the smooth coefficient uncertainty matrix
  # to use for the entire forecast horizon of this particular path
  gam_coef_index <- sample(seq(1, NROW(coef_sim)), 1, T)
  # For each following timestep, recursively predict based on the
  # predictions at each previous lag
  for (t in 3:(h + 2)) {
    # Build the GAM linear predictor matrix using the two previous lags
    # of the (log) density
    newdata <- data.frame(lag1 = log(states[t-1] + 0.01),
                          lag2 = log(states[t-2] + 0.01),
                          season = lynx_test$season[t-2],
                          year = lynx_test$year[t-2])
    colnames(newdata) <- c('lag1', 'lag2', 'season', 'year')
    Xp <- predict(model, newdata = newdata, type = 'lpmatrix')
    # Calculate the posterior prediction for this timepoint
    mu <- rpois(1, lambda = exp(Xp %*% coef_sim[gam_coef_index,]))
    # Fill in the state vector and iterate to the next timepoint
    states[t] <- mu
  }
  # Return the forecast path
  states[-c(1:2)]
}
```

Create the GAM's forecast distribution by generating `1000` simulated forecast paths. Each path is fed the true observed values for the last two lags of the first out of sample timepoint, but they can deviate when simulating ahead depending on their particular draw of possible coefficients. Note, this is a bit slow and could easily be parallelised to speed up computations

```
gam_sims <- matrix(NA, nrow = 1000, ncol = 10)
for(i in 1:1000){
  gam_sims[i,] <- recurse_nonlin(lynx_mgcv,
                                 coef_sim,
                                 lagged_vals = c(lynx_test$lag1[1],
                                                 lynx_test$lag2[1]),
                                 h = 10)
}
```

Plot the mgcv model's out of sample forecast for the next 10 years ahead

```
cred_ints <- apply(gam_sims, 2, function(x) quantile(x, probs = c(0.025, 0.5, 0.975)))
yupper <- max(cred_ints) * 1.25
plot(cred_ints[3,] ~ seq(1:NCOL(cred_ints)), type = 'l',
     col = rgb(1,0,0, alpha = 0),
     ylim = c(0, yupper),
     ylab = 'Predicted lynx trappings',
     xlab = 'Forecast horizon',
     main = 'mgcv')
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
        c(cred_ints[1,],rev(cred_ints[3,])),
        col = rgb(150, 0, 0, max = 255, alpha = 100), border = NA)
cred_ints <- apply(gam_sims, 2, function(x) quantile(x, probs = c(0.15, 0.5, 0.85)))
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
        c(cred_ints[1,],rev(cred_ints[3,])),
        col = rgb(150, 0, 0, max = 255, alpha = 180), border = NA)
lines(cred_ints[2,], col = rgb(150, 0, 0, max = 255), lwd = 2, lty = 'dashed')
points(lynx_test$population[1:10], pch = 16)
lines(lynx_test$population[1:10])
```



A decent forecast? The shape is certainly correct, but the 95% uncertainty intervals appear to be far too wide (i.e. our upper interval extends to up to ~8 times the maximum number of trappings that have ever been recorded up to this point). This is almost entirely due to the extrapolation behaviour of the B spline, as the lag smooth functions are not encountering values very far outside the ranges they've already been trained on so they are resorting mostly to interpolation. But a better way to evaluate than simply using visuals is to calculate a probabilistic score. Here we use the Discrete Rank Probabiilty Score, which gives us an indication of how well calibrated our forecast's uncertainty intervals are by comparing the mass of the forecast density against the true observed values. Forecasts with overly wide intervals are penalised, as are forecasts with overly narrow intervals that do not contain the true observations. At the same time we calculate coverage of the forecast's 90% intervals, which is another useful way of evaluating different forecast proposals

Click for definition of `drps_score`

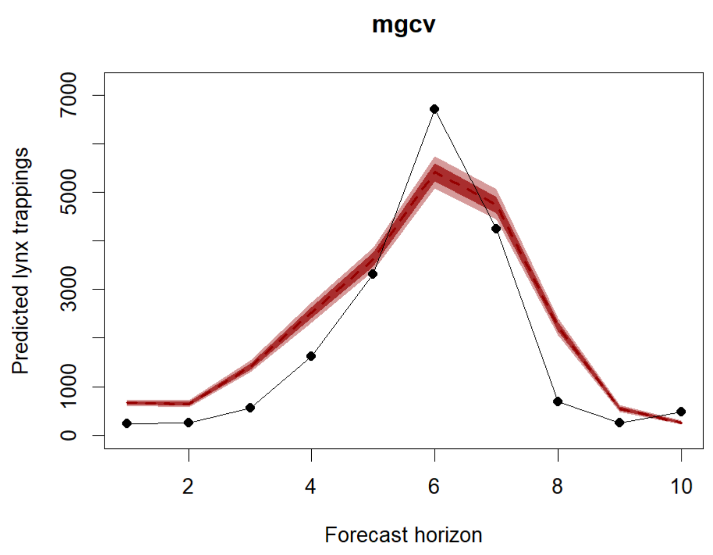
```
# Discrete Rank Probability Score and coverage of 90% interval
drps_score <- function(truth, fc, interval_width = 0.9){
  nsum <- 1000
  Fy = ecdf(fc)
  ysum <- 0:nsum
  indicator <- ifelse(ysum - truth >= 0, 1, 0)
  score <- sum((indicator - Fy(ysum))^2)

  # Is value within 90% HPD?
  interval <- quantile(fc, probs = c(((1-0.9)/2), 0.5, 1-((1-interval_width)/2)))
  in_interval <- ifelse(truth <= interval[3] & truth >= interval[1], 1, 0)
  return(c(score, in_interval))
}

# Wrapper to operate on all observations in fc_horizon
drps_mcmc_object <- function(truth, fc, interval_width = 0.9){
  indices_keep <- which(!is.na(truth))
  if(length(indices_keep) == 0){
    scores = data.frame('drps' = rep(NA, length(truth)),
                        'interval' = rep(NA, length(truth)))
  } else {
    scores <- matrix(NA, nrow = length(truth), ncol = 2)
    for(i in indices_keep){
      scores[i,] <- drps_score(truth = as.vector(truth)[i],
                                fc = fc[,i], interval_width)
    }
  }
  scores
}
```

Calculate DRPS over the 10-year horizon for the mgcv model

```
lynx_mgcv1_drps <- drps_mcmc_object(truth = lynx_test$population[1:10],
                   fc = gam_sims)
```

What if we remove the yearly trend and let the lag smooths capture more of the temporal dependencies? Will that improve the forecast distribution? Run a second model and plot the forecast (note that this plot will be on quite a different y-axis scale compared to the first plot above)

```
lynx_mgcv2 = gam(population ~
                  s(season, bs = 'cc', k = 19) +
                  s(lag1, k = 5) +
                  s(lag2, k = 5),
                data = lynx_train, family = "poisson",
                method = "REML")
coef_sim <- gam.mh(lynx_mgcv2)$bs
gam_sims <- matrix(NA, nrow = 1000, ncol = 10)
for(i in 1:1000){
  gam_sims[i,] <- recurse_nonlin(lynx_mgcv2,
                                coef_sim,
                                lagged_vals = c(lynx_test$lag1[1],
                                                lynx_test$lag2[1]),
                                h = 10)
}
cred_ints <- apply(gam_sims, 2, function(x) quantile(x, probs = c(0.025, 0.5, 0.975)))
yupper <- max(cred_ints) * 1.25
plot(cred_ints[3,] ~ seq(1:NCOL(cred_ints)), type = 'l',
     col = rgb(1,0,0, alpha = 0),
     ylim = c(0, yupper),
     ylab = 'Predicted lynx trappings',
     xlab = 'Forecast horizon',
     main = 'mgcv')
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
        c(cred_ints[1,],rev(cred_ints[3,])),
        col = rgb(150, 0, 0, max = 255, alpha = 100), border = NA)
cred_ints <- apply(gam_sims, 2, function(x) quantile(x, probs = c(0.15, 0.5, 0.85)))
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
        c(cred_ints[1,],rev(cred_ints[3,])),
        col = rgb(150, 0, 0, max = 255, alpha = 180), border = NA)
lines(cred_ints[2,], col = rgb(150, 0, 0, max = 255), lwd = 2, lty = 'dashed')
points(lynx_test$population[1:10], pch = 16)
lines(lynx_test$population[1:10])
```



Calculate DRPS over the 10-year horizon for the second mgcv model

```
lynx_mgcv2_drps <- drps_mcmc_object(truth = lynx_test$population[1:10],
                                   fc = gam_sims)
```

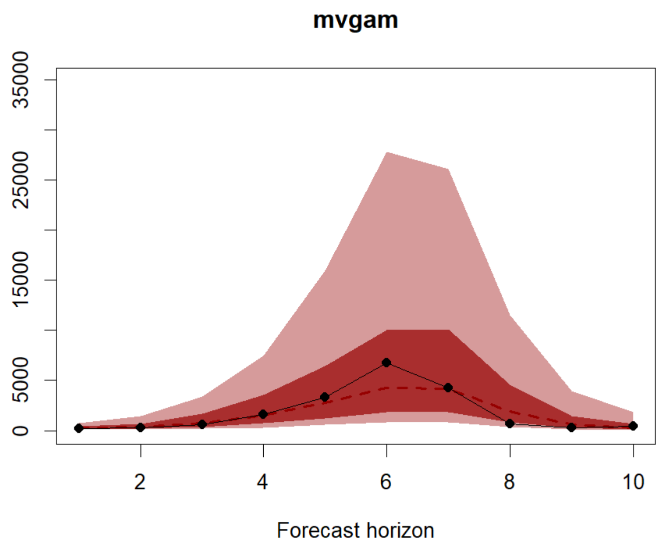
This forecast is highly overconfident, with very unrealistic uncertainty intervals due to the interpolation behaviours of the lag smooths. You can certainly keep trying different formulations (our experience is that the B spline variant above produces the best forecasts from any tested `mgcv` model, but we did not test an exhaustive set), but hopefully it is clear that forecasting using splines is tricky business and it is likely that each time you do it you'll end up honing in on different combinations of penalties, knot selections etc.... Now we will fit an `mvgam` model for comparison. This model fits a similar model to the `mgcv` model directly above but with a full time series model for the errors (in this case an `AR1` process), rather than smoothing splines that do not incorporate a concept of the future. We do not use a `year` term to reduce any possible extrapolation and because the latent dynamic component should capture this temporal variation.

```
lynx_mvgam <- mvgam(data = lynx_train,
               newdata = lynx_test,
               formula = y ~ s(season, bs = 'cc', k = 19),
               knots = list(season = c(0.5, 19.5)),
               family = 'poisson',
               trend_model = 'AR1',
               use_stan = TRUE,
               chains = 4)
```

```
## Running MCMC with 4 parallel chains...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 2 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 4 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 3 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 2 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 4 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 3 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 4 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 2 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 3 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 4 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 2 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 3 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 4 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 2 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 3 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 3 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 4 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 4 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 2 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 2 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 3 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 4 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 2 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 3 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 4 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 2 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 3 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 2 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 4 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 3 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 13.9 seconds.
## Chain 2 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 4 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 4 finished in 14.6 seconds.
## Chain 3 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 3 finished in 15.0 seconds.
## Chain 2 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 2 finished in 15.9 seconds.
##
## All 4 chains finished successfully.
## Mean chain execution time: 14.8 seconds.
## Total execution time: 16.0 seconds.
```

Calculate the out of sample forecast from the fitted `mvgam` model and plot

```r
fits <- MCMCvis::MCMCchains(lynx_mvgam$model_output, 'ypred')
fits <- fits[,(NROW(lynx_mvgam$obs_data)+1):(NROW(lynx_mvgam$obs_data)+10)]
cred_ints <- apply(fits, 2, function(x) quantile(x, probs = c(0.025, 0.5, 0.975)))
yupper <- max(cred_ints) * 1.25
plot(cred_ints[3,] ~ seq(1:NCOL(cred_ints)), type = 'l',
     col = rgb(1,0,0, alpha = 0),
     ylim = c(0, yupper),
     ylab = '',
     xlab = 'Forecast horizon',
     main = 'mvgam')
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
        c(cred_ints[1,],rev(cred_ints[3,])),
        col = rgb(150, 0, 0, max = 255, alpha = 100), border = NA)
cred_ints <- apply(fits, 2, function(x) quantile(x, probs = c(0.15, 0.5, 0.85)))
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
        c(cred_ints[1,],rev(cred_ints[3,])),
        col = rgb(150, 0, 0, max = 255, alpha = 180), border = NA)
lines(cred_ints[2,], col = rgb(150, 0, 0, max = 255), lwd = 2, lty = 'dashed')
points(lynx_test$population[1:10], pch = 16)
lines(lynx_test$population[1:10])
```



Calculate DRPS over the 10-year horizon for the mvgam model

```r
lynx_mvgam_drps <- drps_mcmc_object(truth = lynx_test$population[1:10],
                                   fc = fits)
```

How do the out of sample DRPS scores stack up for these three models? Remember, our goal is to minimise DRPS while providing 90% intervals that are near, but not less than, `0.9`. The DRPS and 90% interval coverage for the first `mgcv` model (with the `B` spline year term)

```r
sum(lynx_mgcv1_drps[,1])
```

```
## [1] 1562.001
```

```r
mean(lynx_mgcv1_drps[,2])
```

```
## [1] 0.7
```

For the second `mgcv` model

```r
sum(lynx_mgcv2_drps[,1])
```

```
## [1] 2037.688
```

```r
mean(lynx_mgcv2_drps[,2])
```

```
## [1] 0
```

And for the `mvgam` model

```r
sum(lynx_mvgam_drps[,1])
```

```
## [1] 808.6506
```

```r
mean(lynx_mvgam_drps[,2])
```

```
## [1] 1
```

The `mvgam` has much more realistic uncertainty than the `mgcv` versions above. Of course this is just one out of sample comparison, and to really determine which model is most appropriate for forecasting we would want to run many of these tests using a rolling window approach. Have a look at this model's summary to see what is being estimated (note that longer MCMC runs would probably be needed to increase effective sample sizes)

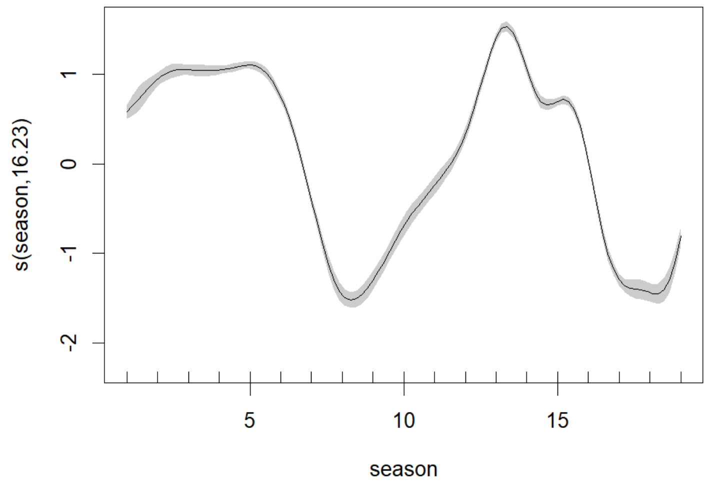
```r
summary(lynx_mvgam)
```

```
## GAM formula:

## y ~ s(season, bs = "cc", k = 19)

##

## Family:

## Poisson

##

## Link function:

## log

##

## Trend model:

## AR1

##

## N series:

## 1

##

## N observations:

## 40

##

## Status:

## Fitted using Stan

##

## GAM coefficient (beta) estimates:
```

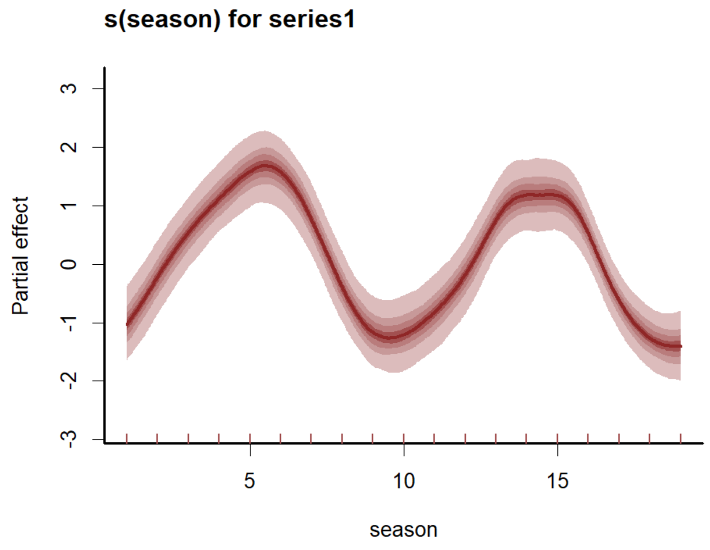
```
##                   2.5%        50%       97.5% Rhat n.eff
## (Intercept)   6.7563450  6.7756600  6.79568000 1.00  2297
## s(season).1  -1.3345672 -0.6860385 -0.01813143 1.00   963
## s(season).2  -0.4322225  0.2602270  0.90884045 1.00   948
## s(season).3   0.3144628  1.0342500  1.75811200 1.01   945
## s(season).4   0.8840928  1.6291650  2.37504375 1.01   763
## s(season).5   1.1271142  1.9628100  2.69864000 1.01   824
## s(season).6   0.6093458  1.3991850  2.15310525 1.00   769
## s(season).7  -0.6648989  0.1414005  0.95776417 1.00   717
## s(season).8  -1.6000782 -0.8328910 -0.01078634 1.00   778
## s(season).9  -1.7145375 -0.9226770 -0.07582052 1.00   640
## s(season).10 -1.3565890 -0.4808305  0.35178922 1.00   650
## s(season).11 -0.5846837  0.2958715  1.14096600 1.01   706
## s(season).12  0.4328673  1.2769500  2.13901475 1.00   703
## s(season).13  0.6484345  1.4637950  2.28755025 1.00   869
## s(season).14  0.4320416  1.2306550  1.94834700 1.00  1084
## s(season).15 -0.5156658  0.0833561  0.69633672 1.00  1161
## s(season).16 -1.3021137 -0.7243125 -0.14791630 1.00  1179
## s(season).17 -1.6816863 -1.1239050 -0.56215133 1.00   841

##

## GAM smoothing parameter (rho) estimates:

##               2.5%     50%    97.5% Rhat n.eff
## s(season) 3.274759 3.95406 4.457864    1  1698

##

## Latent trend parameter estimates:

##               2.5%       50%     97.5% Rhat n.eff
## ar1[1]    0.5424553 0.7676760 0.9758150    1  1587
## sigma[1]  0.3609405 0.4579115 0.6032218    1   998

##

## Stan MCMC diagnostics

## n_eff / iter looks reasonable for all parameters
## Rhat looks reasonable for all parameters
## 0 of 2000 iterations ended with a divergence (0%)
## 0 of 2000 iterations saturated the maximum tree depth of 12 (0%)
## E-FMI indicated no pathological behavior

##
```

Now inspect each model's estimated smooth for the 19-year cyclic pattern. Note that the `mvgam` smooth plot is on a different scale compared to the `mgcv` plot, but interpretation is similar. The `mgcv` smooth is much wigglier, likely because it is compensating for any remaining autocorrelation not captured by the lag smooths. We could probably remedy this by reducing `k` in the seasonal smooth for the `mgcv` model (in practice this works well, but leaving `k` larger for the `mvgam`'s seasonal smooth is recommended as our experience is that this tends to lead to better performance and convergence)

```
plot(lynx_mgcv, select=1, shade=T)
```



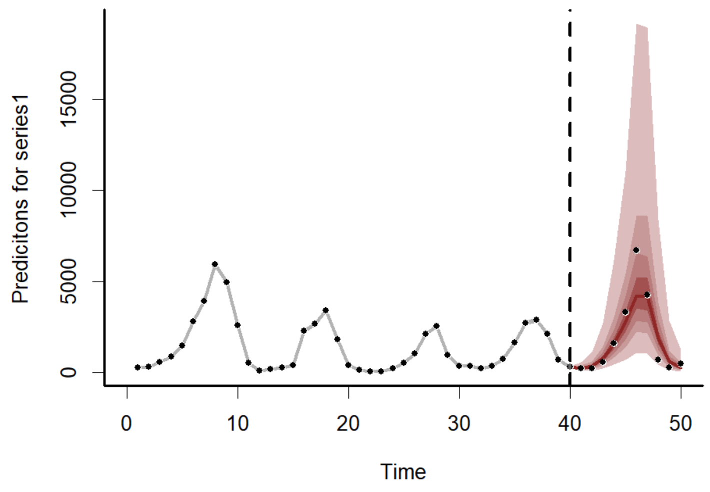
```
plot_mvgam_smooth(lynx_mvgam, 1, 'season')
```



We can also view the mvgam's posterior predictions for the entire series (testing and training)

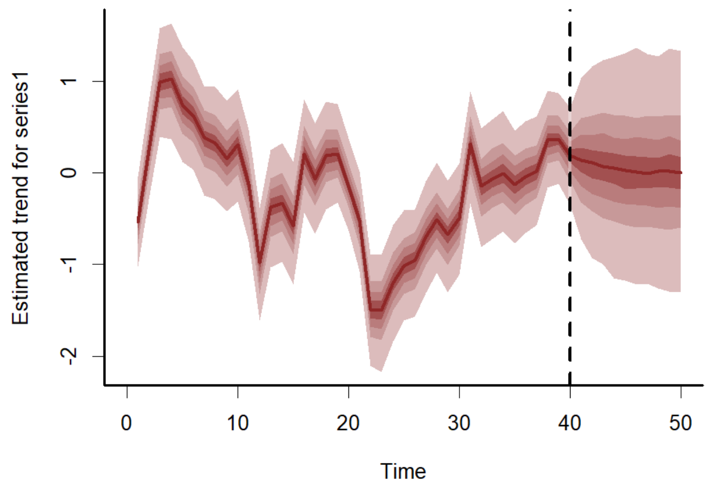
```
plot(lynx_mvgam, type = 'forecast', newdata = lynx_test)
```

```
## Out of sample DRPS:

## [1] 808.6506

##
```



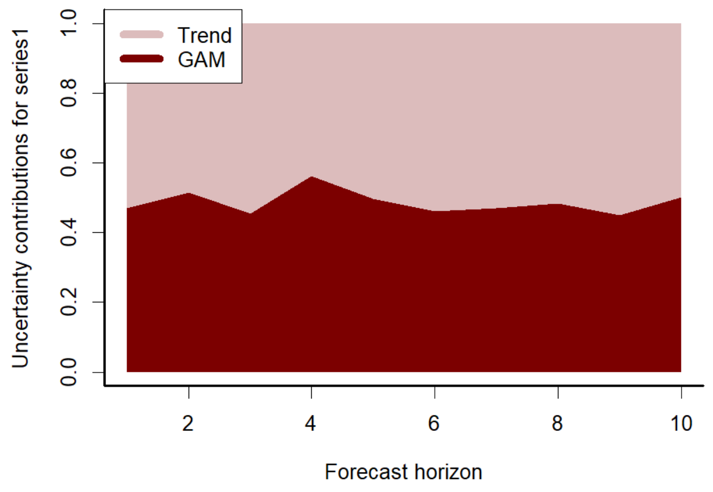
And the estimated trend

```r
plot(lynx_mvgam, type = 'trend', newdata = lynx_test)
```



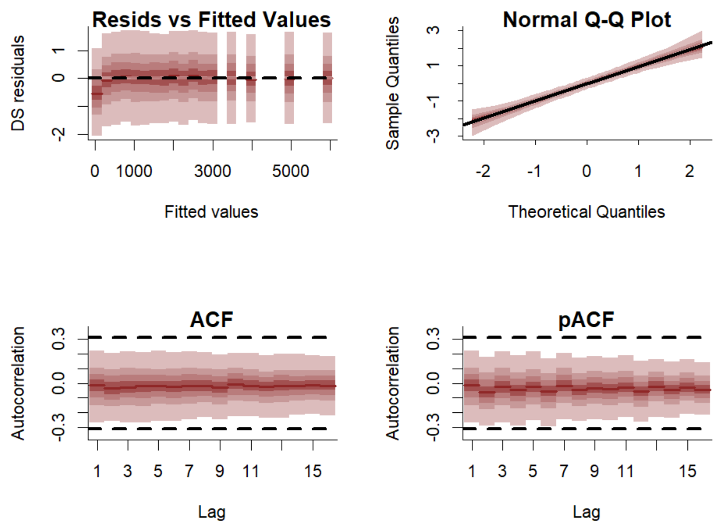
A key aspect of ecological forecasting is to understand how different components of a model contribute to forecast uncertainty. We can estimate contributions to forecast uncertainty for the GAM smooth functions and the latent trend using `mvgam`

```r
plot(lynx_mvgam, type = 'uncertainty', newdata = lynx_test)
```



Both components contribute to forecast uncertainty, with the trend component contributing more over time (as it should since this is the stochastic forecast component). This suggests we would still need some more work to learn about factors driving the dynamics of the system. But we will leave the model as-is for this example. Diagnostics of the model can also be performed using `mvgam`. Have a look at the model's Dunn-Smyth randomised quantile residuals. Again we have more options for plotting when using the `plot_mvgam_resids()` function compared to the `S3` generic `plot.mvgam()`. Here we use this flexibility to modify the number of bins for the residuals vs fitted plot (top-left) so that the patterns are more clear. We are primarily looking for a lack of autocorrelation, which would suggest our AR1 model is appropriate for the latent trend

```
plot_mvgam_resids(lynx_mvgam, n_bins = 25)
```



References

Clark, N.J. and Wells, K. (2022). Dynamic Generalized Additive Models (DGAMs) for forecasting discrete ecological time series. *Methods in Ecology and Evolution.* DOI: https://doi.org/10.1111/2041-210X.13974