mvgam case study 1: model comparison and data assimilation

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

Generalised Additive Models (GAMs) are incredibly flexible tools that have found particular application in the analysis of time series. In ecology, a host of recent papers and workshops (i.e. the 2018 Ecological Society of America workshop on GAMs (https://noamross.github.io/mgcv-esa-2018/) that was hosted by Eric Pedersen, David L. Miller, Gavin Simpson, and Noam Ross) have drawn special attention to the wide range of applications that GAMs have for addressing complex ecological problems. Given the many ways that GAMs can model temporal data, it is tempting to use the smooth functions estimated by a GAM to produce out of sample forecasts. Here we will inspect the behaviours of smoothing splines when extrapolating to data outside the range of the training data to examine whether this can be useful in practice.

We will work in the `mvgam` package, which fits dynamic GAMs (DGAMs) using MCMC sampling via the `JAGS` software (Note that `JAGS` is required; installation links are found here (https://sourceforge.net/projects/mcmc-jags/files/)). Briefly, assume $\tilde{\boldsymbol{y}}_t$ is the conditional expectation of a discrete response variable $\boldsymbol{y}$ at time $\boldsymbol{t}$. Asssuming $\boldsymbol{y}$ 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 the covariates ($\boldsymbol{x}$'s) and $\boldsymbol{z}$ is a dynamic latent trend component. Each smooth function $\boldsymbol{s}_i$ is composed of spline like basis expansions whose coefficients, which must be estimated, control the shape of the functional relationship between $\boldsymbol{x}_i$ and $log(\tilde{\boldsymbol{y}})$. The size of the basis expansion limits the smooth's potential complexity, with a larger set of basis functions allowing greater flexibility. Several advantages of GAMs are that they can model a diversity of response families, including discrete distributions (i.e. Poisson or Negative Binomial) that accommodate ecological features such as zero-inflation, 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?

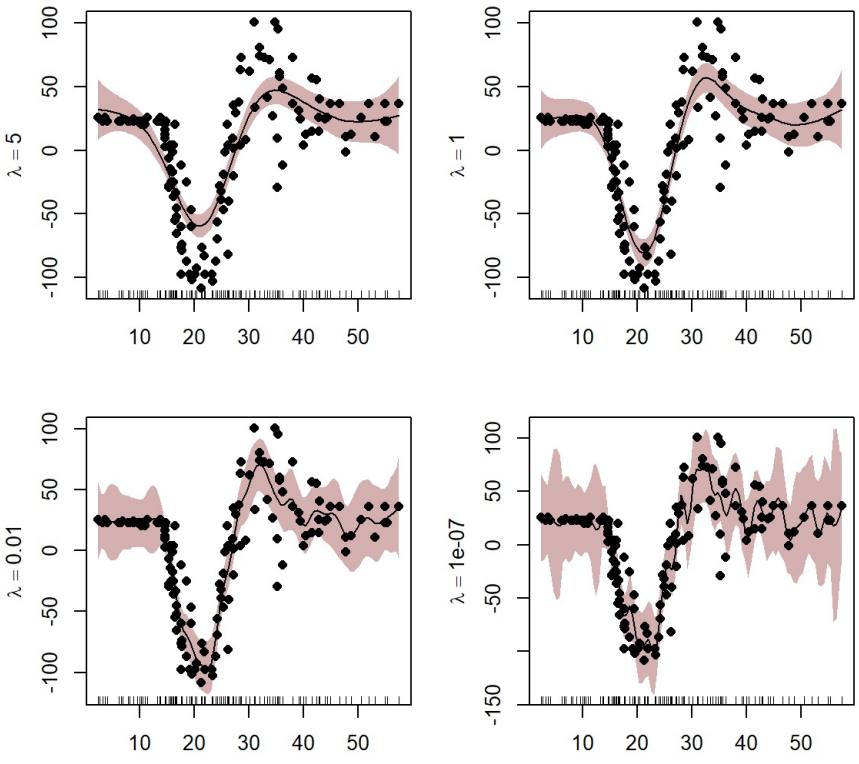
Dynamic GAMs are useful when we wish to predict future values from time series that show temporal dependence and we want to avoid extrapolating from a smooth (which, as you'll see below, can sometimes lead to unpredictable and unrealistic behaviours). In addition, smooths can often try to wiggle excessively to capture any autocorrelation that is present 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 large value
par(mfrow = c(2, 2), mgp = c(2.5, 1, 0),
    mai = c(0.6, 0.6, 0.2, 0.2))
library(mgcv)
```

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

```
## This is mgcv 1.8-33. For overview type 'help("mgcv-package")'.
```

```
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,
    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,
    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,
    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 = 1e-07)
plot(m4, scheme = 1, residuals = TRUE, pch = 16,
    ylab = expression(lambda == 1e-07), 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, as the function will need to have a very high degree of flexibility in order to model both the true function and the autocorrelation

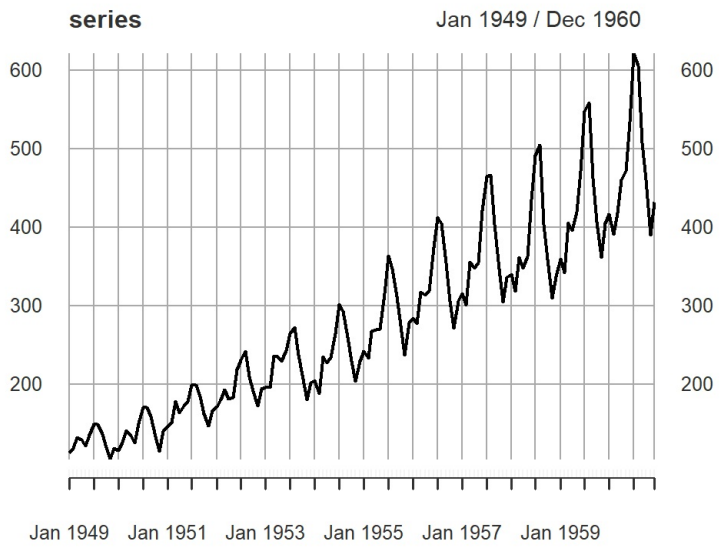
AirPassengers example

Now onto an empirical example. First we load the `AirPassengers` data from the `forecast` package and convert to an `xts` object. This series is a good starting point as it should be highly forecastable given its stable seasonal pattern and nearly linear trend

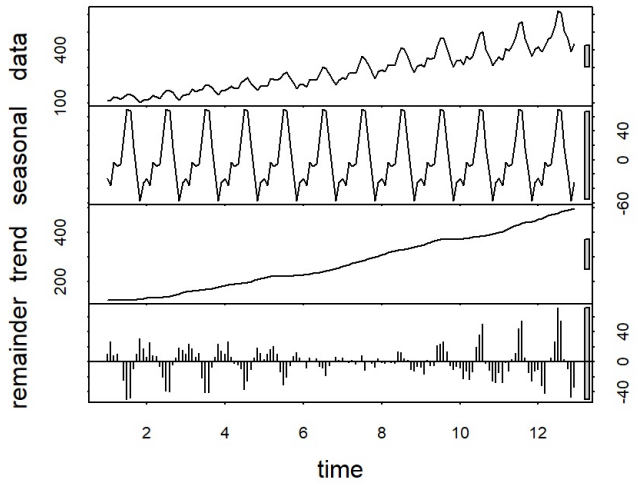
```
# devtools::install_github('nicholasjclark/mvgam')
library(mvgam)
library(dplyr)
library(xts)
library(forecast)
data("AirPassengers")
series <- xts::as.xts(floor(AirPassengers))
colnames(series) <- c("Air")
```

View the raw series, its STL decomposition and the distribution of observations. There is a clear seasonal pattern as well as an increasing trend over time for this series, and the distribution shows evidence of a skew suggestive of overdispersion

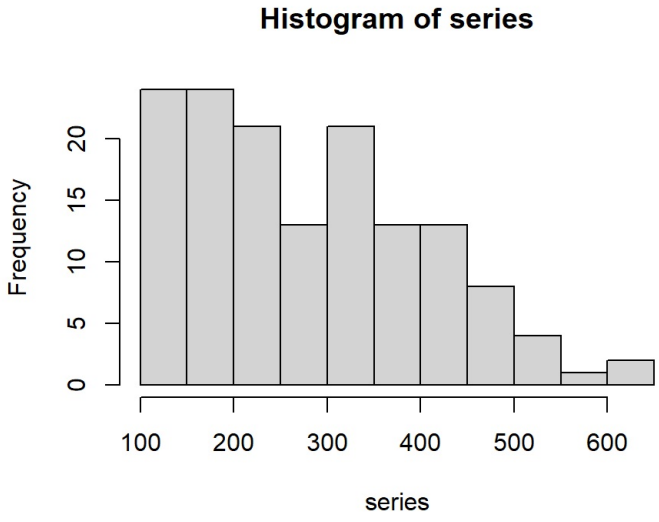
```
plot(series)
```




```
plot(stl(series, s.window = "periodic"))
```




```
hist(series)
```



Next use the `series_to_mvgam` function, which converts `ts` or `xts` objects into the correct format for `mvgam`. Here we set `train_prop = 0.75`, meaning we will include ~ 75% of the observations in the training set and use the remaining ~25% for forecast validation. We also randomly set ~10% of observations to `NA` so that we can evaluate models based on their predictive abilities

```
series[sample(1:length(series), floor(length(series)/10),
    F)] <- NA
fake_data <- series_to_mvgam(series, freq = 12,
    train_prop = 0.75)
```

Examine the returned object

```
head(fake_data$data_train)
```

1
2
3
4
5
6

6 rows | 1-1 of 7 columns

```
head(fake_data$data_test)
```

1
2
3
4
5
6

6 rows | 1-1 of 7 columns

As a first pass at modelling this series, we will fit a GAM that includes a seasonal smooth and a yearly smooth, as well as their tensor product interaction. As the seasonality is cyclic, we will use a cyclic cubic regression spline for `season`. The knots are set so that the boundaries of the cyclic smooth match up between December 31 and January 1. We will stick with the default thin plate regression spline for `year`. This is similar to what we might do when fitting a model in `mgcv` to try and forecast ahead, except here we also have an explicit model for the residual component. `mvgam` uses the `jagam` function from the `mgcv` package to generate a skeleton `JAGS` file and updates that file to incorporate any dynamic trend components (so far limited to no trend, random walks or AR models up to order 3). This is advantageous as any GAM that is allowed in `mgcv` can in principle be used in `mvgam` to fit dynamic linear models with smooth functions for nonlinear covariate effects. For multivariate series, `mvgam` also includes an option for dimension reduction by inducing trend correlations via a dynamic factor process. Here we use the Negative Binomial

family and a burnin length of `10000` iterations (in practice we would probably run these models for a bit longer, but they tend to converge rather quickly for univariate series thanks to the useful starting values supplied by `jagam`). Note that feeding the `data_test` object does not mean that these values are used in training of the model. Rather, they are included as `NA` so that we can automatically create a forecast from the posterior predictions for these observations. This is useful for plotting forecasts without needing to run new data through the model's equations later on

```
mod1 <- mvjagam(data_train = fake_data$data_train,
    data_test = fake_data$data_test, formula = y ~
        s(season, bs = c("cc"), k = 12) +
            s(year, k = 5) + ti(season, year,
            bs = c("cc", "tp"), k = c(12,
                3)), knots = list(season = c(0.5,
        12.5)), family = "nb", trend_model = "None",
    burnin = 1000, chains = 4)
```

```
## NOTE: Stopping adaptation
```

We can view the `JAGS` model file that has been generated to see the additions that have been made to the base `gam` model. If the user selects `return_jags_data = TRUE` when calling `mvjagam`, this file can be modified and the resulting `jags_data` object can also be modified to fit more complex Bayesian models. Note that here the AR and phi terms have been set to zero as this model does not include a dynamic trend component

```
mod1$model_file
```

```
##  [1] "model {"
##  [2] ""
##  [3] "## GAM linear predictor"
##  [4] "eta <- X %*% b"
##  [5] ""
##  [6] "## Mean expectations"
##  [7] "for (i in 1:n) {"
##  [8] " for (s in 1:n_series) {"
##  [9] "  mu[i, s] <- exp(eta[ytimes[i, s]] + trend[i, s])"
## [10] " }"
## [11] "}"
## [12] ""
## [13] ""
## [14] "## State space trend estimates"
## [15] "for(s in 1:n_series) {"
## [16] " trend[1, s] <- 0"
## [17] "}"
## [18] ""
## [19] "for(s in 1:n_series) {"
## [20] " trend[2, s] <- 0"
## [21] "}"
## [22] ""
## [23] "for(s in 1:n_series) {"
## [24] " trend[3, s] <- 0"
## [25] "}"
## [26] ""
## [27] "for (i in 4:n){"
## [28] " for (s in 1:n_series){"
## [29] " trend[i, s] <- 0"
## [30] " }"
## [31] "}"
## [32] ""
## [33] "## AR components"
## [34] "for (s in 1:n_series){"
## [35] " phi[s] <- 0"
## [36] " ar1[s] <- 0"
## [37] " ar2[s] <- 0"
## [38] " ar3[s] <- 0"
## [39] " tau[s] <- pow(sigma[s], -2)"
## [40] " sigma[s] ~ dexp(1)"
## [41] "}"
## [42] ""
## [43] "## Negative binomial likelihood functions"
## [44] "for (i in 1:n) {"
## [45] " for (s in 1:n_series) {"
## [46] "  y[i, s] ~ dnegbin(rate[i, s], r[s])"
## [47] "  rate[i, s] <- ifelse((r[s] / (r[s] + mu[i, s])) < min_eps, min_eps,"
## [48] "                       (r[s] / (r[s] + mu[i, s])))"
## [49] " }"
## [50] "}"
## [51] ""
## [52] "## Complexity penalising prior for the overdispersion parameter;"
## [53] "## where the likelihood reduces to a 'base' model (Poisson) unless"
```

```
## [54] "## the data support overdispersion"
## [55] "for(s in 1:n_series){"
## [56] " r[s] <- pow(r_raw[s], 2)"
## [57] " r_raw[s] ~ dexp(0.05)"
## [58] "}"
## [59] ""
## [60] ""
## [61] "## Posterior predictions"
## [62] "for (i in 1:n) {"
## [63] " for (s in 1:n_series) {"
## [64] "  ypred[i, s] ~ dnegbin(rate[i, s], r[s])"
## [65] " }"
## [66] "}"
## [67] ""
## [68] "  ## parametric effect priors (regularised for identifiability)"
## [69] "  for (i in 1:1) { b[i] ~ dnorm(p_coefs[i], p_taus[i]) }"
## [70] "  ## prior for s(season)... "
## [71] "  K1 <- S1[1:10,1:10] * lambda[1] "
## [72] "  b[2:11] ~ dmnorm(zero[2:11],K1) "
## [73] "  ## prior for s(year)... "
## [74] "  K2 <- S2[1:4,1:4] * lambda[2]  + S2[1:4,5:8] * lambda[3]"
## [75] "  b[12:15] ~ dmnorm(zero[12:15],K2) "
## [76] "  ## prior for ti(season,year)... "
## [77] "  K3 <- S3[1:20,1:20] * lambda[4]  + S3[1:20,21:40] * lambda[5]"
## [78] "  b[16:35] ~ dmnorm(zero[16:35],K3) "
## [79] "   ## smoothing parameter priors..."
## [80] "  for (i in 1:5) {"
## [81] "   lambda[i] ~ dexp(0.05)"
## [82] "    rho[i] <- log(lambda[i])"
## [83] "  }"
## [84] "}"
```

Inspect the summary from the model (`mvgam` includes `S3` generics for `summary()`, `plot()`, `predict()` and `print()`), which is somewhat similar to an `mgcv` model summary with extra information about convergences for unobserved parameters. The estimated degrees of freedom, smooth coefficients and smooth penalties are all extracted from the `mvgam` model using `sim2jam` so that approximate p-values can be calculated using Nychka's method (following Wood (2013) Biometrika 100(1), 221-228). Note however that this function is still in development and approximate p-values may not be entirely accurate

```
summary(mod1)
```

```
## GAM formula:
```

```
## y ~ s(season, bs = c("cc"), k = 12) + s(year, k = 5) + ti(season,
##     year, bs = c("cc", "tp"), k = c(12, 3))
```

```
##
```

```
## Family:
```

```
## Negative Binomial
```

```
##
```

```
## Link function:
```

```
## log
```

```
##
```

```
## Trend model:
```

```
## None
```

```
##
```

```
## N series:
```

```
## 1
```

```
##
```

```
## N observations per series:
```

```
## 108
```

```
##
```

```
## Status:
```

```
## Fitted using runjags::run.jags()
```

```
##
```

```
## Dispersion parameter estimates:
```

```
##       2.5%      50%    97.5% Rhat n.eff
## r 1368.008 4854.402 19555.22    1  6436
```

```
##
```

```
## GAM smooth term approximate significances:
```

```
##                    edf Ref.df Chi.sq p-value
## s(season)        9.262 10.000 732.70  <2e-16 ***
## s(year)          2.888  4.000  12.33  0.0027 **
## ti(season,year) 12.368 20.000  13.11  0.4014
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
```

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

```
##                           2.5%          50%       97.5% Rhat n.eff
## (Intercept)          5.360853979  5.375014671  5.38956935 1.00  5938
## s(season).1         -0.289802508 -0.212331501 -0.13798432 1.01   318
## s(season).2         -0.155722907 -0.106860622 -0.05853394 1.00   669
## s(season).3         -0.043632229  0.012527355  0.06911542 1.03   500
## s(season).4         -0.070025316 -0.022376027  0.02156731 1.01   838
## s(season).5          0.019744384  0.068060818  0.11362856 1.02   813
## s(season).6          0.173447135  0.219826556  0.26555265 1.01   774
## s(season).7          0.154354793  0.197027279  0.23800210 1.01   895
## s(season).8         -0.001033037  0.059153321  0.12428600 1.02   345
## s(season).9         -0.203261947 -0.149211091 -0.09908083 1.07   629
## s(season).10        -0.200940619 -0.125250137 -0.04947633 1.03   319
## s(year).1           -0.094460213 -0.025702935  0.03272985 1.06   256
## s(year).2           -0.135303747 -0.010074675  0.11026357 1.19    76
## s(year).3           -0.127648500  0.011939546  0.15766829 1.19    73
## s(year).4            0.315045983  0.368654227  0.42984559 1.03   294
## ti(season,year).1   -0.053846894  0.041100055  0.14400050 1.00   300
## ti(season,year).2   -0.118791115 -0.025924183  0.06701422 1.01   289
## ti(season,year).3   -0.039281080  0.059600731  0.15478360 1.04   286
## ti(season,year).4   -0.133807199 -0.047826440  0.03320101 1.01   355
## ti(season,year).5   -0.082231525  0.020255553  0.11675970 1.01   295
## ti(season,year).6   -0.112811248 -0.028323082  0.05945958 1.03   380
## ti(season,year).7   -0.108765172 -0.015872254  0.06973388 1.03   337
## ti(season,year).8   -0.090949845  0.003439215  0.09625515 1.02   332
## ti(season,year).9   -0.125554525 -0.032510143  0.05537361 1.03   325
## ti(season,year).10  -0.038901245  0.038797534  0.11418706 1.01   432
## ti(season,year).11  -0.123828409 -0.031963465  0.05455795 1.01   332
## ti(season,year).12  -0.021623483  0.047970917  0.12226022 1.01   455
## ti(season,year).13  -0.110598614 -0.022931715  0.05809624 1.02   392
## ti(season,year).14  -0.021977018  0.044457877  0.11266234 1.02   558
## ti(season,year).15  -0.106949232  0.006479484  0.12878628 1.08   202
## ti(season,year).16  -0.084469548  0.003359900  0.08180599 1.03   371
## ti(season,year).17  -0.096113260 -0.002210237  0.08673426 1.04   335
## ti(season,year).18  -0.088342005 -0.005298547  0.07950141 1.02   357
## ti(season,year).19  -0.087380101  0.010358327  0.10978728 1.01   305
## ti(season,year).20  -0.113733938 -0.020429591  0.06317577 1.01   323
```

```
##
```

```
## GAM smoothing parameter (rho) estimates:
```

```
##                        2.5%      50%    97.5% Rhat n.eff
## s(season)         3.36115210 4.358189 5.158743 1.01  2134
## s(year)           1.59787419 3.345391 4.549377 1.01  1653
## s(year)2         -0.07696683 2.298393 3.655027 1.00  7863
## ti(season,year)   3.59691282 4.577892 5.334329 1.00  3116
## ti(season,year)2  2.49656793 3.861247 4.852769 1.00  1889
```

```
##
```

```
## Latent trend drift (phi) and AR parameter estimates:
```

```
##     2.5% 50% 97.5% Rhat n.eff
## phi    0   0     0  NaN     0
## ar1    0   0     0  NaN     0
## ar2    0   0     0  NaN     0
## ar3    0   0     0  NaN     0
```

```
##
```

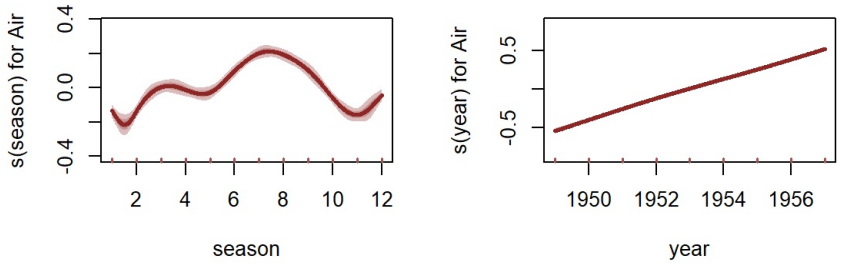
`mvgam` takes a fitted `gam` model and adapts the model file to fit in `JAGS` , with possible extensions to deal with stochastic trend components and other features. But as this model has not been modified much from the original `gam` model with the same formula (i.e. we have not added any stochastic trend components to the linear predictor yet, which is the main feature of `mvgam`), the summary of the unmodified `gam` model is also useful and fairly accurate

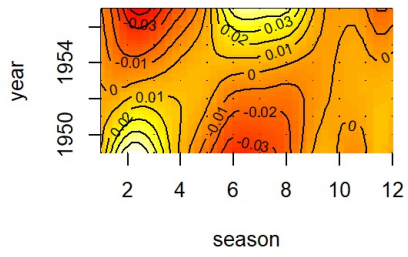
```
summary(mod1$mgcv_model)
```

```
##
## Family: Negative Binomial(64119502.489)
## Link function: log
##
## Formula:
## y ~ s(season, bs = c("cc"), k = 12) + s(year, k = 5) + ti(season,
##     year, bs = c("cc", "tp"), k = c(12, 3))
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 5.378931   0.007162     751   <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)       7.789     10  312.954  <2e-16 ***
## s(year)         1.000      1 2444.420  <2e-16 ***
## ti(season,year) 1.599     20    4.644  0.0371 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.988   Deviance explained = 98.8%
## -REML =  392.8  Scale est. = 1         n = 96
```

Ordinarily we would be quite pleased with this result, as we have explained most of the variation in the series with a fairly simple model. We can plot the estimated smooth functions and their associated credible intervals, which are interpreted similarly to `mgcv` plots with the exception of the partial residuals. `mvgam` uses Dunn-Smyth residuals that are drawn from a standard normal distribution rather than working or pearson residuals as in `mgcv`

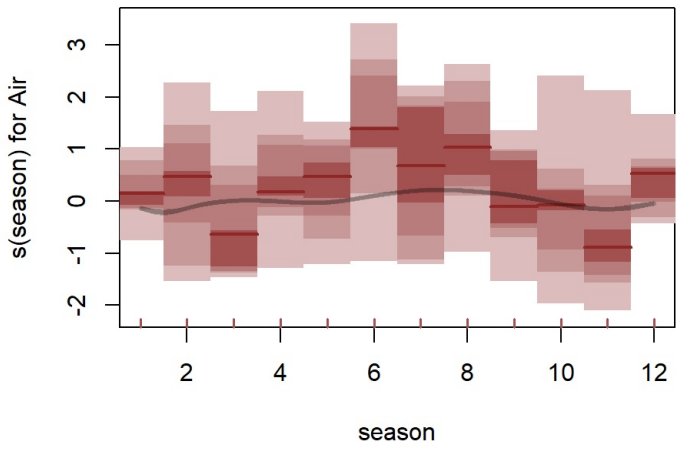
```
plot(mod1, series = 1, type = "smooths")
```





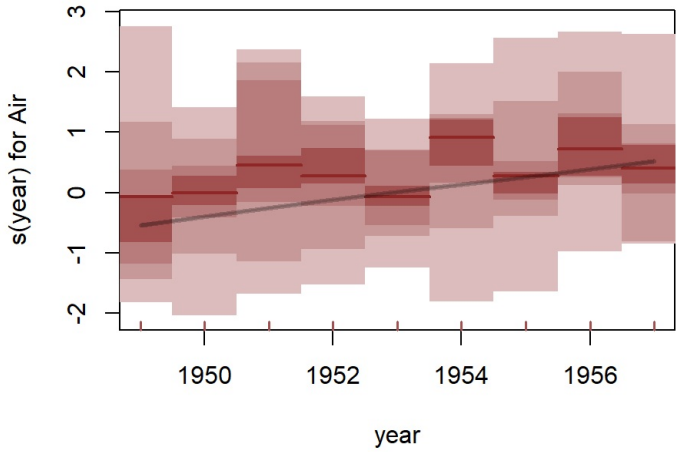
The `plot_mvgam_smooth` function allows more flexibility for plotting smooth functions, including an ability to supply `newdata` for plotting posterior marginal simulations. Overlay the partial Dunn-Smyth residuals on the smooth plot for `season`

```
plot_mvgam_smooth(mod1, series = 1, smooth = "season",
    residuals = TRUE)
```



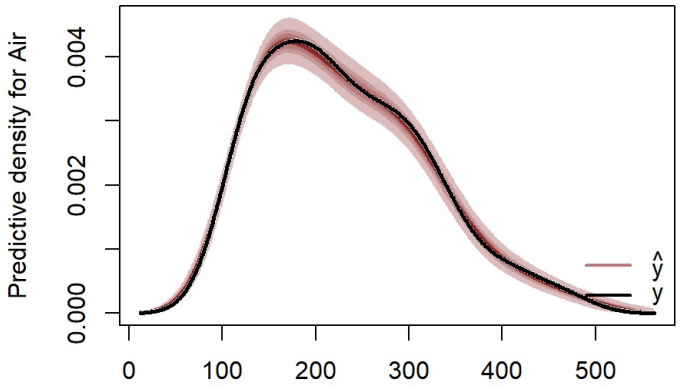
And here is the plot of the smooth function for `year`, which has essentially estimated a straight line

```
plot_mvgam_smooth(mod1, series = 1, smooth = "year",
    residuals = TRUE)
```



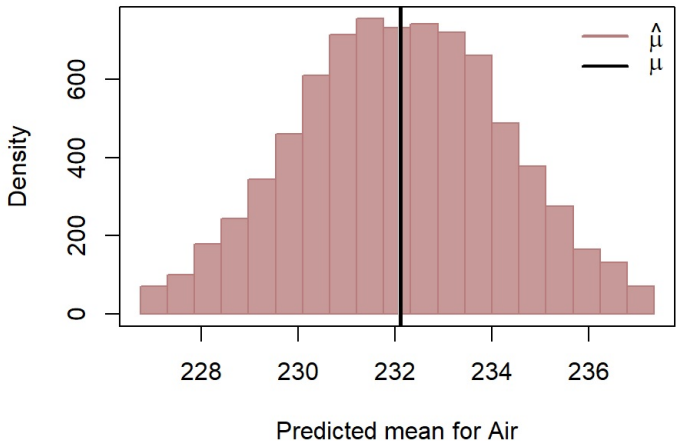
Perform a series of posterior predictive checks (using the `ppc()` function) to see if the model is able to simulate data for the training period that looks realistic and unbiased. First, examine simulated kernel densities for posterior predictions (`yhat`) and compare to the density of the observations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

```
ppc(mod1, series = 1, type = "density", legend_position = "bottomright")
```



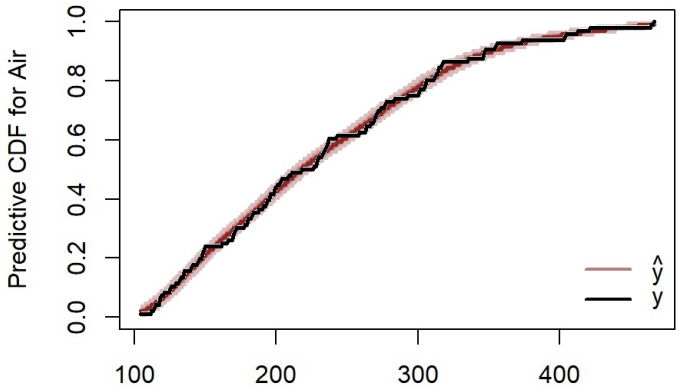
Now plot the distribution of predicted means compared to the observed mean

```
ppc(mod1, series = 1, type = "mean")
```



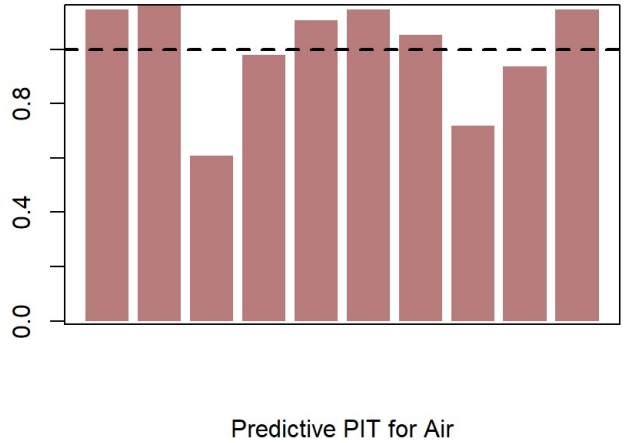
Next examine simulated empirical Cumulative Distribution Functions (CDF) for posterior predictions (`yhat`) and compare to the CDF of the observations (`y`)

```
ppc(mod1, series = 1, type = "cdf")
```



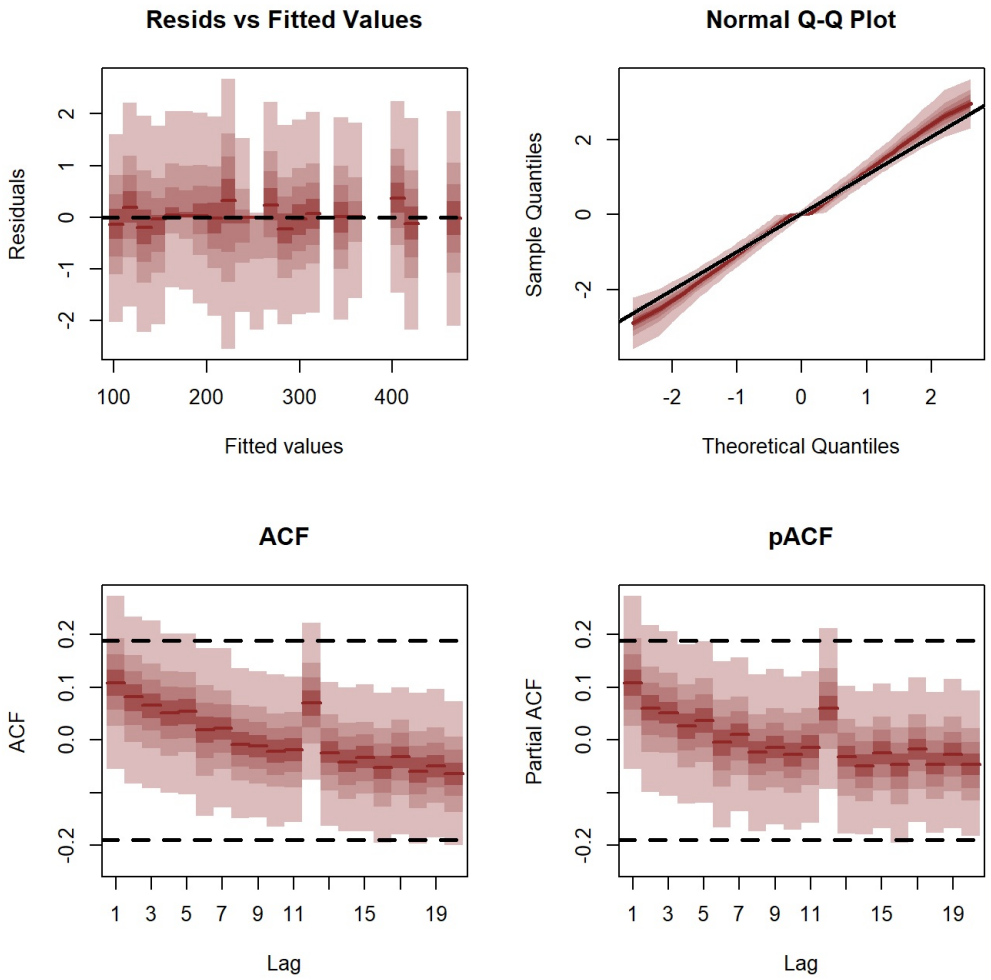
Finally look for any biases in predictions by examining a Probability Integral Transform (PIT) histogram. If our predictions are not biased one way or another (i.e. not consistently under- or over-predicting), this histogram should look roughly uniform

```
ppc(mod1, series = 1, type = "pit")
```



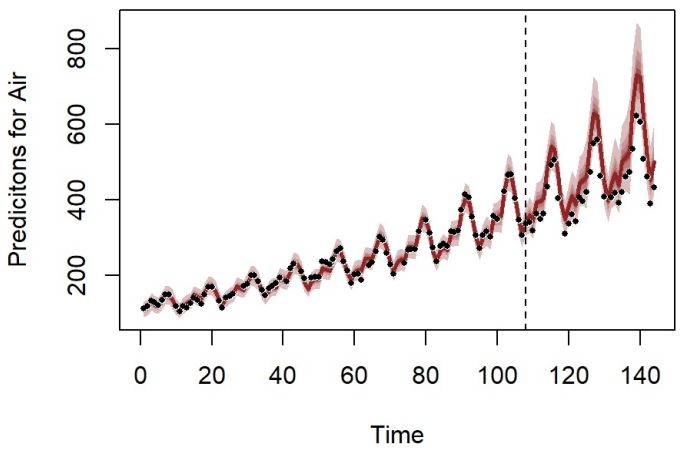
All of these plots indicate the model is well calibrated against the training data, with no apparent pathological behaviors exhibited. Now for some investigation of the estimated relationships and forecasts. We can also perform residual diagnostics using randomised quantile (Dunn-Smyth) residuals. These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

```
plot(mod1, series = 1, type = "residuals")
```



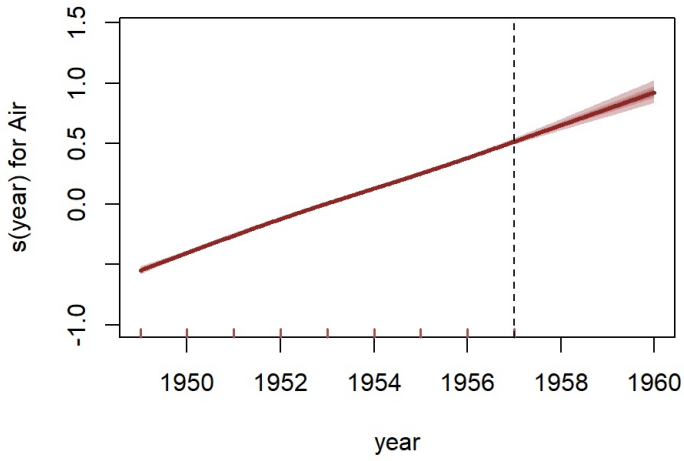
Ok so the model is doing well when fitting against the training data, but how are its forecasts? The yearly trend is being extrapolated into the future, which controls most of the shape and uncertainty in the forecast. We see there is a reasonable estimate of uncertainty and the out of sample observations (to the right of the dashed line) are all within the model's 95% HPD intervals

```
plot(mod1, series = 1, type = "forecast",
    data_test = fake_data$data_test)
```



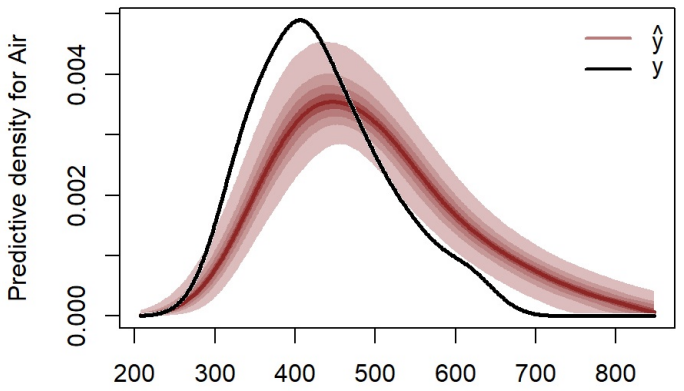
The extrapolation of the smooth for `year` can be better viewed by feeding new data to the more flexible `plot_mvgam_smooth()` function. Here we feed values of `year` to cover the training and testing set to see how the extrapolation would continue into the future. The dashed line marks the end of the training period

```
plot_mvgam_smooth(mod1, series = 1, "year",
    newdata = expand.grid(year = seq(min(fake_data$data_train$year),
        max(fake_data$data_test$year), length.out = 500),
        season = mean(fake_data$data_train$season),
        series = unique(fake_data$data_train$series)))
abline(v = max(fake_data$data_train$year),
    lty = "dashed")
```

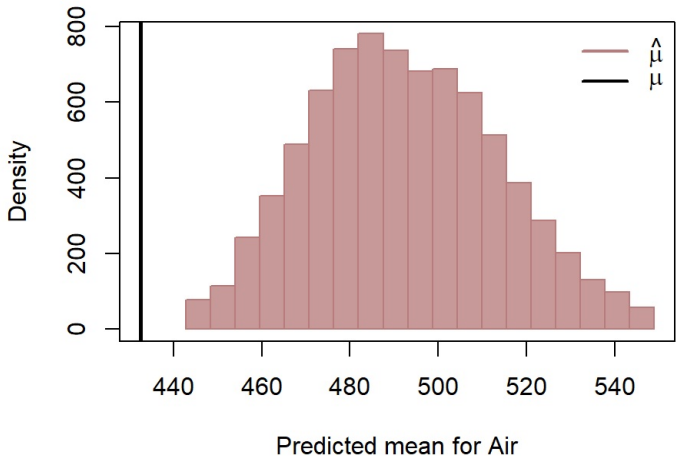


We can also re-do the posterior predictive checks, but this time focusing only on the out of sample period. This will give us better insight into how the model is performing and whether it is able to simulate realistic and unbiased future values

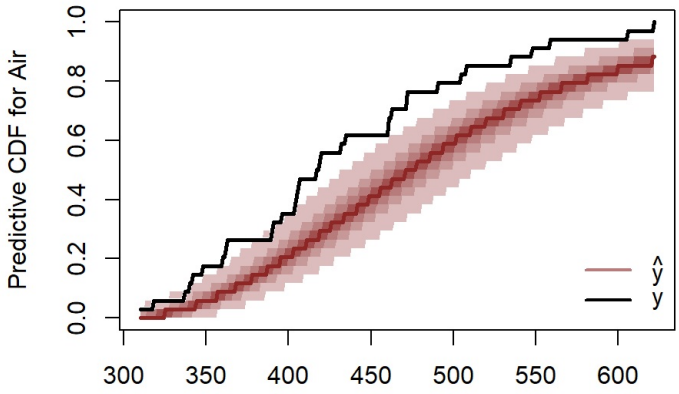
```
ppc(mod1, series = 1, type = "density", data_test = fake_data$data_test,
    legend_position = "topright")
```



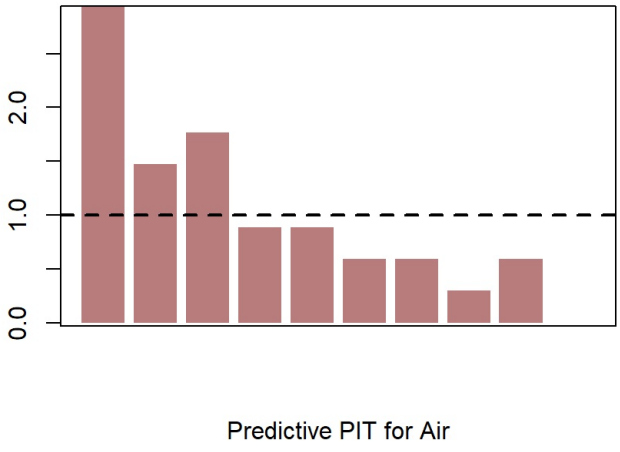

```
ppc(mod1, series = 1, type = "mean", data_test = fake_data$data_test)
```




```
ppc(mod1, series = 1, type = "cdf", data_test = fake_data$data_test)
```




```
ppc(mod1, series = 1, type = "pit", data_test = fake_data$data_test)
```



There are some problems with the way this model is generating future predictions. Perhaps a different smooth function for `year` can help? Here we fit our original model again but use a different smooth term for `year` to try and capture the long-term trend (using `B` splines with multiple penalties, following the excellent example by Gavin Simpson about extrapolating with smooth terms (https://fromthebottomoftheheap.net/2020/06/03/extrapolating-with-gams/)). 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`). Note that we drop the `year*season` tensor product interaction to help emphasize the extrapolation behaviour of the `B` spline

```
mod2 <- mvjagam(data_train = fake_data$data_train,
    data_test = fake_data$data_test, formula = y ~
        s(season, bs = c("cc"), k = 12) +
            s(year, bs = "bs", m = c(2, 1)),
    knots = list(season = c(0.5, 12.5), year = c(min(fake_data$data_train$year) -
        1, min(fake_data$data_train$year),
        max(fake_data$data_train$year), max(fake_data$data_test$year))),
    family = "nb", trend_model = "None",
    burnin = 10000, chains = 4)
```

```
## NOTE: Stopping adaptation
```

Again as we haven't modified the base `gam` much, the summary from the `mgcv` model is fairly accurate

```
summary(mod2)
```

```
## GAM formula:
```

```
## y ~ s(season, bs = c("cc"), k = 12) + s(year, bs = "bs", m = c(2,
##     1))
```

```
##
```

```
## Family:
```

```
## Negative Binomial
```

```
##
```

```
## Link function:
```

```
## log
```

```
##
```

```
## Trend model:
```

```
## None
```

```
##
```

```
## N series:
```

```
## 1
```

```
##
```

```
## N observations per series:
```

```
## 108
```

```
##
```

```
## Status:
```

```
## Fitted using runjags::run.jags()
```

```
##
```

```
## Dispersion parameter estimates:
```

```
##       2.5%      50%    97.5% Rhat n.eff
## r 1431.696 4872.448 19495.78    1  5621
```

```
##
```

```
## GAM smooth term approximate significances:
```

```
##              edf Ref.df Chi.sq p-value
## s(season)  7.934 10.000  436.2  <2e-16 ***
## s(year)    6.706  9.000 1710.1  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
```

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

```
##                     2.5%          50%        97.5% Rhat n.eff
## (Intercept)   5.36007272  5.374438557  5.389116620 1.00  5605
## s(season).1  -0.21819248 -0.158777116 -0.109750786 1.00   577
## s(season).2  -0.14821406 -0.104348318 -0.062981770 1.00   709
## s(season).3  -0.02749683  0.018765171  0.064276479 1.00   795
## s(season).4  -0.04984593 -0.003003388  0.039857574 1.00   762
## s(season).5   0.04785528  0.088754028  0.128373945 1.00   902
## s(season).6   0.19779386  0.239857758  0.279402402 1.00   847
## s(season).7   0.18014235  0.218753268  0.256417202 1.00   942
## s(season).8   0.02792807  0.071322674  0.114466617 1.01   818
## s(season).9  -0.17077474 -0.120148469 -0.073657594 1.01   606
## s(season).10 -0.15261723 -0.103528649 -0.053934801 1.01   624
## s(year).1    -1.37394854 -0.515139173  0.303181246 1.05   211
## s(year).2    -0.45357025 -0.341866302 -0.233619226 1.00   475
## s(year).3    -0.14598542 -0.074329430 -0.003792237 1.00   957
## s(year).4     0.09648959  0.175869321  0.258427245 1.01   681
## s(year).5     0.16226057  0.231009528  0.300691313 1.00   845
## s(year).6     0.36468171  0.437995722  0.510535086 1.01   753
## s(year).7     0.61054999  0.680125073  0.750227529 1.00   607
## s(year).8     0.52914348  0.697592767  0.857178369 1.00   374
## s(year).9    -0.23823276  0.604186351  1.362030068 1.01   201
```

```
##
```

```
## GAM smoothing parameter (rho) estimates:
```

```
##               2.5%      50%    97.5% Rhat n.eff
## s(season) 5.271312 6.498828 7.529296    1  1210
## s(year)   2.927266 4.069584 4.914291    1  1455
```

```
##
```

```
## Latent trend drift (phi) and AR parameter estimates:
```

```
##     2.5% 50% 97.5% Rhat n.eff
## phi    0   0     0  NaN     0
## ar1    0   0     0  NaN     0
## ar2    0   0     0  NaN     0
## ar3    0   0     0  NaN     0
```

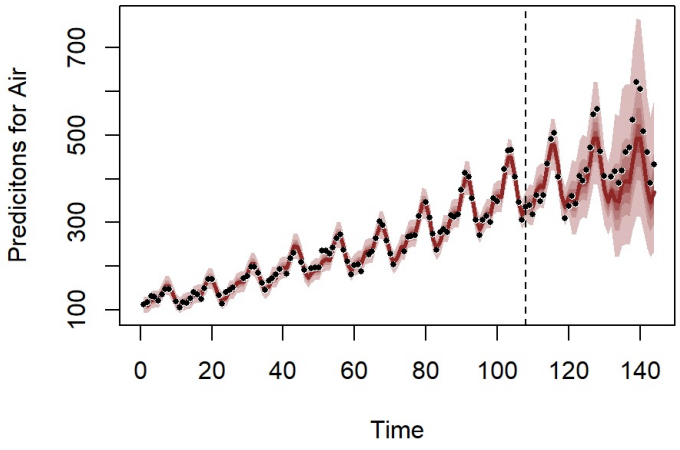
```
##
```

```
summary(mod2$mgcv_model)
```

```
##
## Family: Negative Binomial(33117393.466)
## Link function: log
##
## Formula:
## y ~ s(season, bs = c("cc"), k = 12) + s(year, bs = "bs", m = c(2,
##     1))
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 5.378326   0.007173   749.8   <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) 7.669     10  352.3  <2e-16 ***
## s(year)   6.582      7 2458.6  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.989   Deviance explained =   99%
## -REML = 398.98  Scale est. = 1         n = 96
```

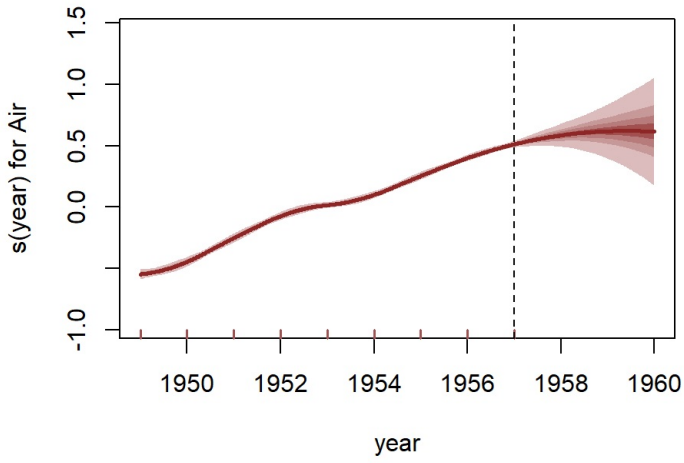
This model explains even more of the variation than the thin plate yearly model above, so we'd be tempted to use it for prediction (though of course we'd want to perform a series of checks following advice from Simon Wood; see lectures by Gavin Simpson for more information on how to perform these checks (https://www.youtube.com/user/ucfagls)). So how do the forecasts look?

```
plot(mod2, series = 1, type = "forecast",
    data_test = fake_data$data_test)
```



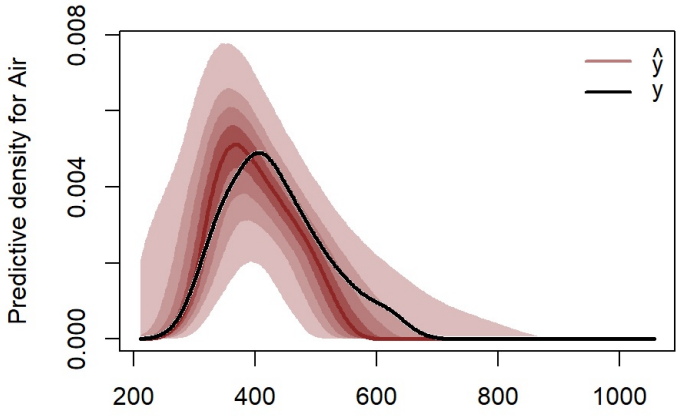
Again the forecast is being driven primarily by the extrapolation behaviour of the `B` spline. Look at this behaviour for the `year` smooth as we did above by feeding new data for prediction

```
plot_mvgam_smooth(mod2, series = 1, "year",
    newdata = expand.grid(year = seq(min(fake_data$data_train$year),
        max(fake_data$data_test$year), length.out = 500),
        season = mean(fake_data$data_train$season),
        series = unique(fake_data$data_train$series)))
abline(v = max(fake_data$data_train$year),
    lty = "dashed")
```

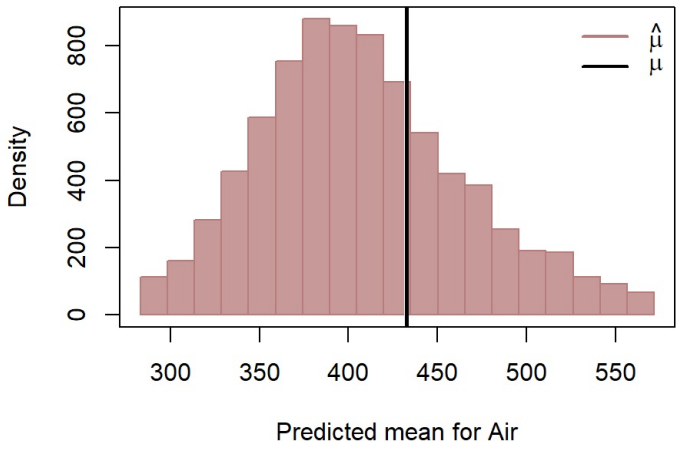


Residual and posterior in-training predictive checks for this model will look similar to the above, with some autocorrelation left in residuals but nothing terribly alarming. But the forecast checks again show some problems:

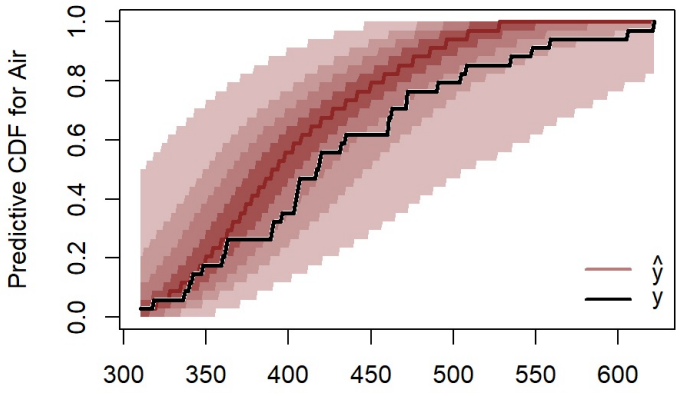
```
ppc(mod2, series = 1, type = "density", data_test = fake_data$data_test)
```



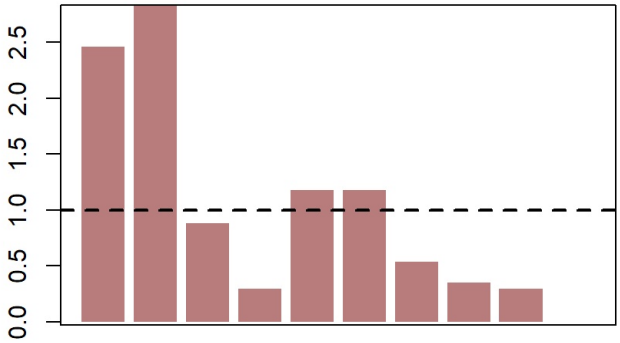

```
ppc(mod2, series = 1, type = "mean", data_test = fake_data$data_test)
```




```
ppc(mod2, series = 1, type = "cdf", data_test = fake_data$data_test)
```




```
ppc(mod2, series = 1, type = "pit", data_test = fake_data$data_test)
```



Predictive PIT for Air

Can we improve the forecasts by removing our reliance on extrapolation? Now we will fit a model in which the GAM component of the linear predictor captures the repeated seasonality (again with a cyclic smooth) and a dynamic latent trend captures the temporal residual process using AR parameters (up to order 3). This model is a mildly modified version of the base `mgcv` model where the linear predictor is augmented with the latent trend component. Slightly longer burnin is used here due to the added complexity of the time series component, but the model still fits in ~ 30 seconds on most machines

```
mod3 <- mvjagam(data_train = fake_data$data_train,
    data_test = fake_data$data_test, formula = y ~
        s(season, bs = c("cc"), k = 12),
    knots = list(season = c(0.5, 12.5)),
    family = "nb", trend_model = "AR3", drift = TRUE,
    burnin = 12000, chains = 4)
```

```
## NOTE: Stopping adaptation
```

In this case the fitted model is more different to the base `mgcv` model that was used in `jagam` to produce the skeleton `JAGS` file, so the summary of that base model is less accurate. But we can still check the model summary for the `mvjagam` mdoel to examine convergence for key parameters

```
summary(mod3)
```

```
## GAM formula:
```

```
## y ~ s(season, bs = c("cc"), k = 12)
```

```
##
```

```
## Family:
```

```
## Negative Binomial
```

```
##
```

```
## Link function:
```

```
## log
```

```
##
```

```
## Trend model:
```

```
## AR3
```

```
##
```

```
## N series:
```

```
## 1
```

```
##
```

```
## N observations per series:
```

```
## 108
```

```
##
```

```
## Status:
```

```
## Fitted using runjags::run.jags()
```

```
##
```

```
## Dispersion parameter estimates:
```

```
##      2.5%      50%    97.5% Rhat n.eff
## r 1364.06 4832.559 19729.05    1  6996
```

```
##
```

```
## GAM smooth term approximate significances:
```

```
##              edf Ref.df Chi.sq p-value
## s(season)  6.546 10.000  352.7  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
```

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

```
##                      2.5%         50%        97.5% Rhat n.eff
## (Intercept)   4.714348678  4.77562410  4.839774581 1.08    43
## s(season).1  -0.128476849 -0.08252154 -0.039511483 1.00   600
## s(season).2  -0.086308558 -0.04319056 -0.002132205 1.01   626
## s(season).3   0.003340551  0.04602177  0.091520211 1.00   721
## s(season).4  -0.024923722  0.02105686  0.063522185 1.02   589
## s(season).5   0.063043263  0.10176050  0.139364969 1.01   758
## s(season).6   0.190942808  0.22961998  0.267747923 1.00   823
## s(season).7   0.164957713  0.20065374  0.236249645 1.00   873
## s(season).8  -0.002132396  0.03893476  0.079010994 1.00   773
## s(season).9  -0.212616022 -0.16607663 -0.121098244 1.01   597
## s(season).10 -0.208739667 -0.16255246 -0.119554187 1.01   683
```

```
##
```

```
## GAM smoothing parameter (rho) estimates:
```

```
##               2.5%      50%    97.5% Rhat n.eff
## s(season) 5.897806 7.123322 8.224316 1.01   992
```

```
##
```

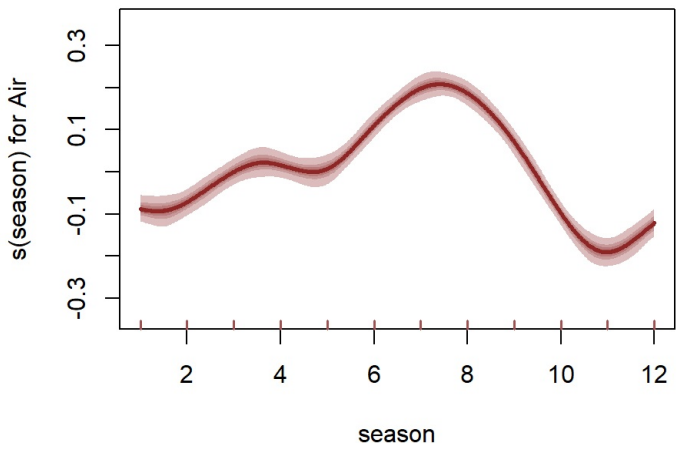
```
## Latent trend drift (phi) and AR parameter estimates:
```

```
##            2.5%        50%      97.5% Rhat n.eff
## phi  0.01238306 0.02303997 0.03404726 1.01   388
## ar1 -0.09585752 0.32284253 0.78080929 1.00  1012
## ar2 -0.16130287 0.31013914 0.72814383 1.01  1514
## ar3 -0.08840037 0.36703258 0.83859896 1.01   772
```

```
##
```

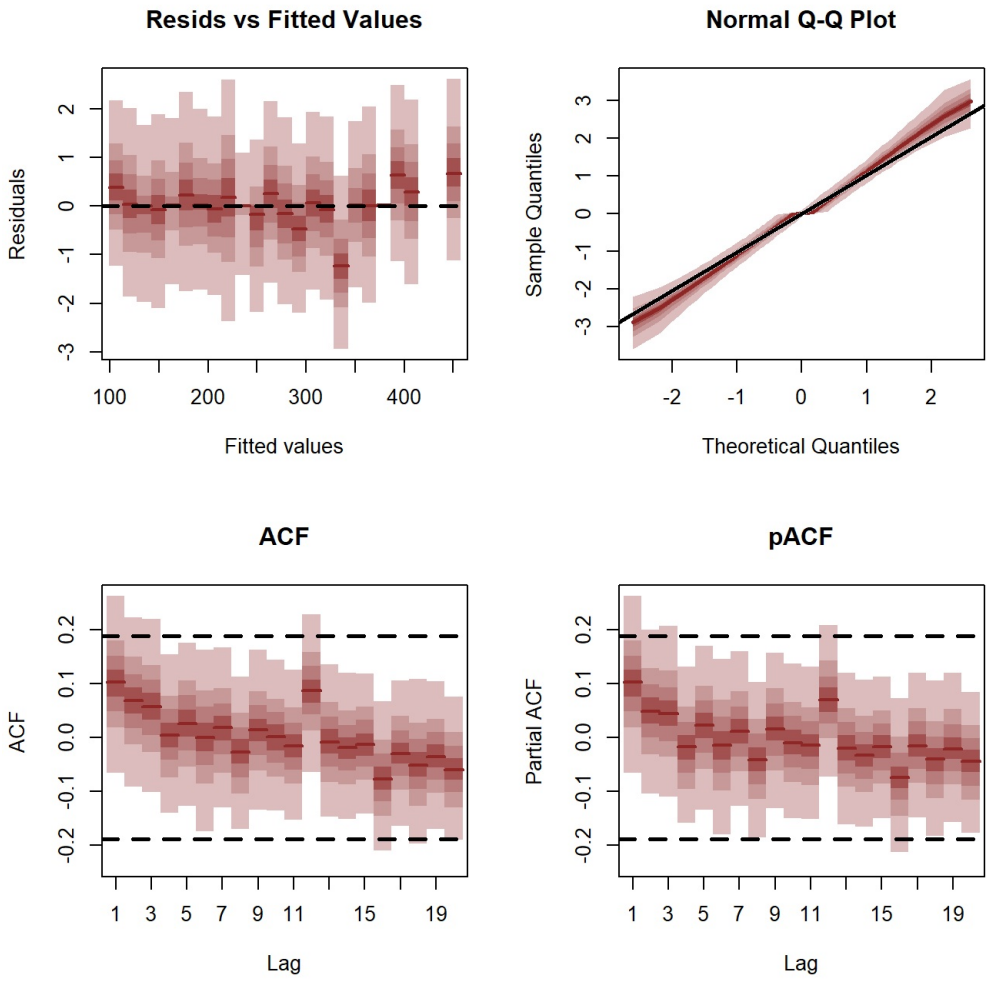
The seasonal term is obviously still very important. Plot it here

```
plot_mvgam_smooth(mod3, series = 1, "season")
```



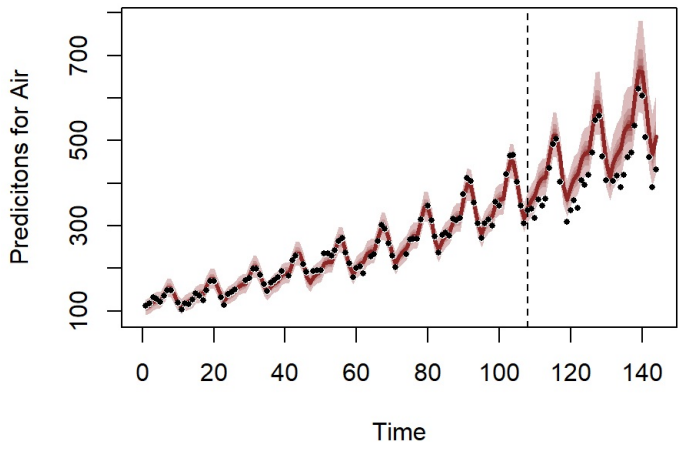
Plot diagnostics of posterior Dunn-Smyth residuals, where the autocorrelation is now captured by the latent trend process

```
plot(mod3, series = 1, type = "residuals")
```

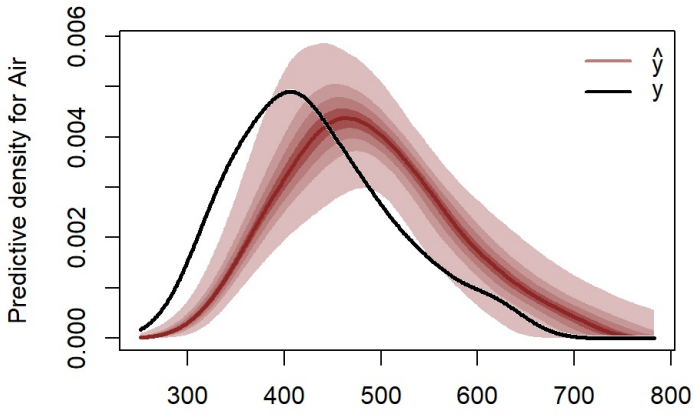


How does the model's posterior forecast distribution compare to the previous models?

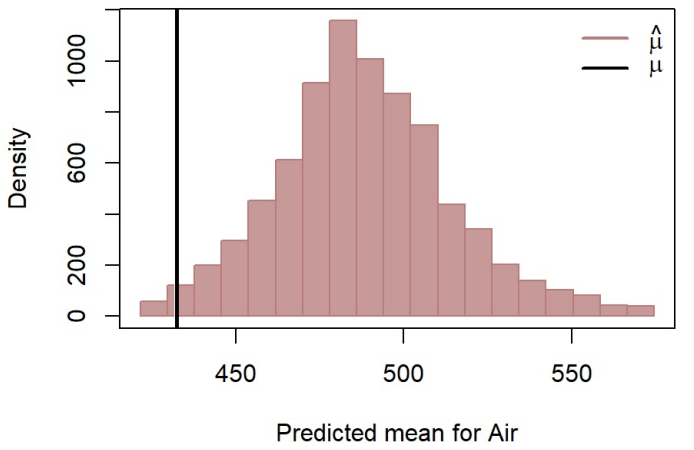
```
plot(mod3, series = 1, type = "forecast",
    data_test = fake_data$data_test)
```



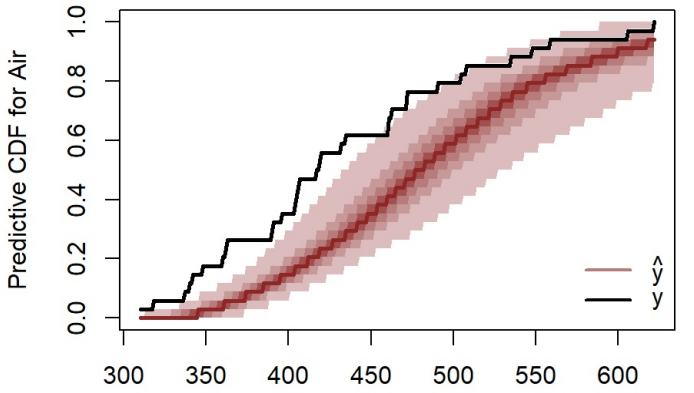

```
ppc(mod3, series = 1, type = "density", data_test = fake_data$data_test)
```



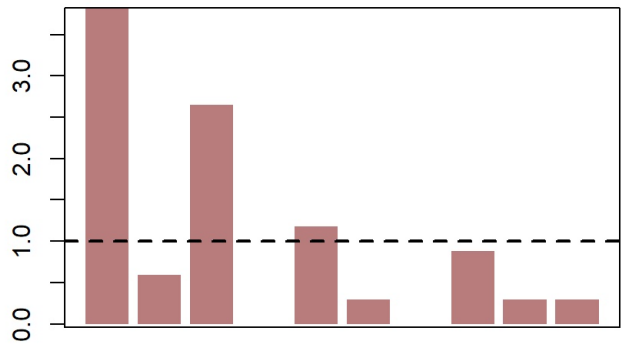

```
ppc(mod3, series = 1, type = "mean", data_test = fake_data$data_test)
```




```
ppc(mod3, series = 1, type = "cdf", data_test = fake_data$data_test)
```



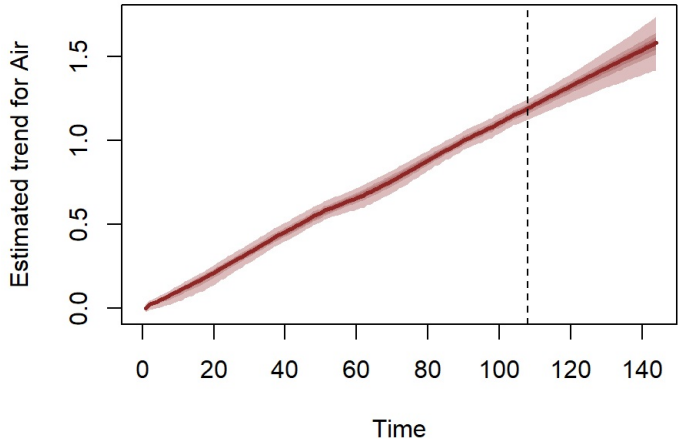

```
ppc(mod3, series = 1, type = "pit", data_test = fake_data$data_test)
```



Predictive PIT for Air

None of the model's is a perfect representation of the data generating process, but the forecast for our dynamic GAM is more stable and more accurate than the previous models, with the added advantage that we can place more trust our estimated smooth for `season` because we have captured the residual autocorrelation. The posterior checks for our dynamic GAM also look much better than the two previous models. Plot the estimated latent dynamic trend (which is on the log scale)

```
plot(mod3, series = 1, type = "trend", data_test = fake_data$data_test)
```



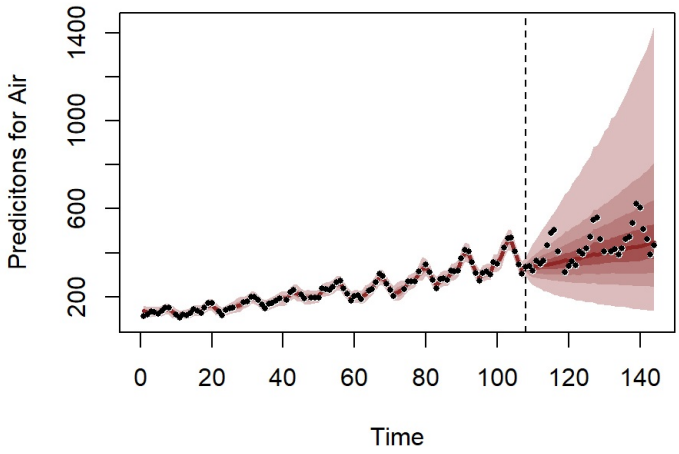
Benchmarking against "null" models is a very important part of evaluating a proposed forecast model. After all, if our complex dynamic model can't generate better predictions then a random walk or mean forecast, is it really telling us anything new about the data-generating process? Here we examine the model comparison utilities in `mvgam` . Here we illustrate how this can be done in `mvgam` by fitting a simpler model by smoothing on a white noise covariate rather than on the seasonal variable. Because the white noise covariate is not informative and we are using a random walk for the trend process, this model essentially becomes a Poisson observation model over a dynamic random walk process.

```
fake_data$data_train$fake_cov <- rnorm(NROW(fake_data$data_train))
fake_data$data_test$fake_cov <- rnorm(NROW(fake_data$data_test))
mod4 <- mvjagam(data_train = fake_data$data_train,
    data_test = fake_data$data_test, formula = y ~
        s(fake_cov, k = 3), family = "poisson",
    trend_model = "RW", drift = TRUE, burnin = 10000,
    chains = 4)
```

```
## NOTE: Stopping adaptation
```

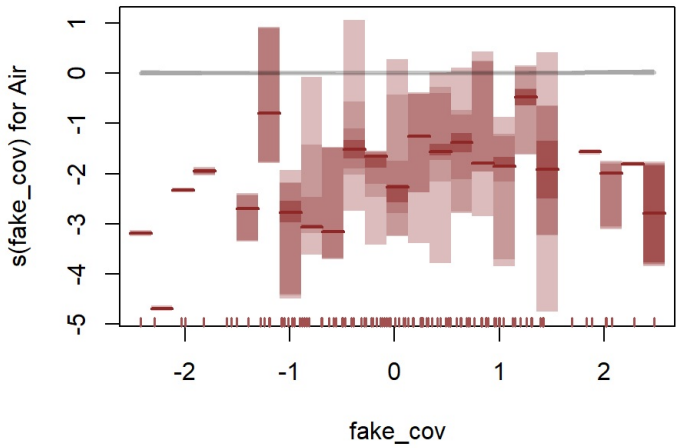
Look at this model's proposed forecast

```
plot(mod4, series = 1, type = "forecast",
    data_test = fake_data$data_test)
```



Inspecting the model's smooth function for the fake covariate `fake_cov` shows that the function is essentially flat and there is no structure in the expected partial residuals, suggesting there would be no stucture remaining if we were to drop this term from the model. Together this evidence indicates that the function is unimportant

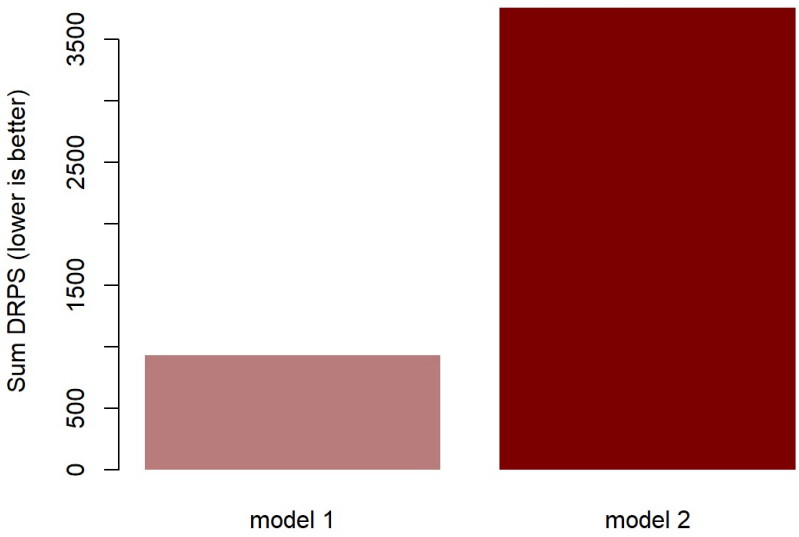
```
plot_mvgam_smooth(mod4, series = 1, smooth = "fake_cov",
    residuals = T)
```

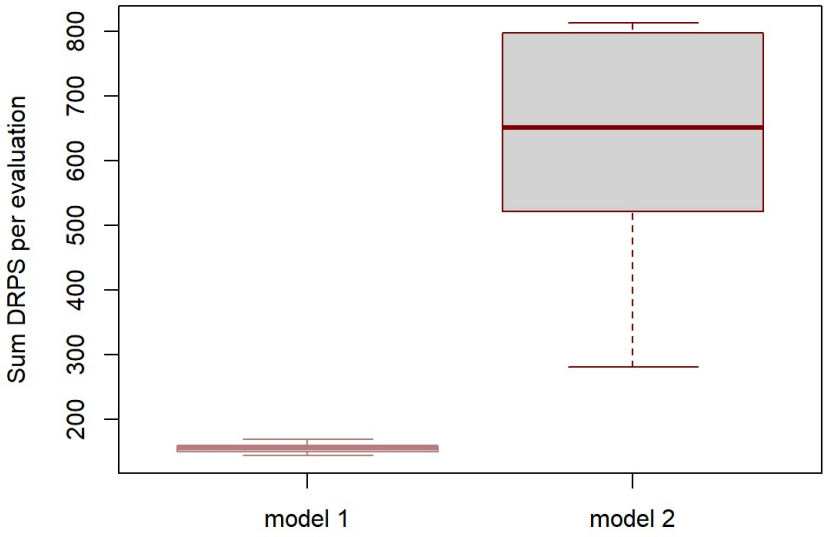


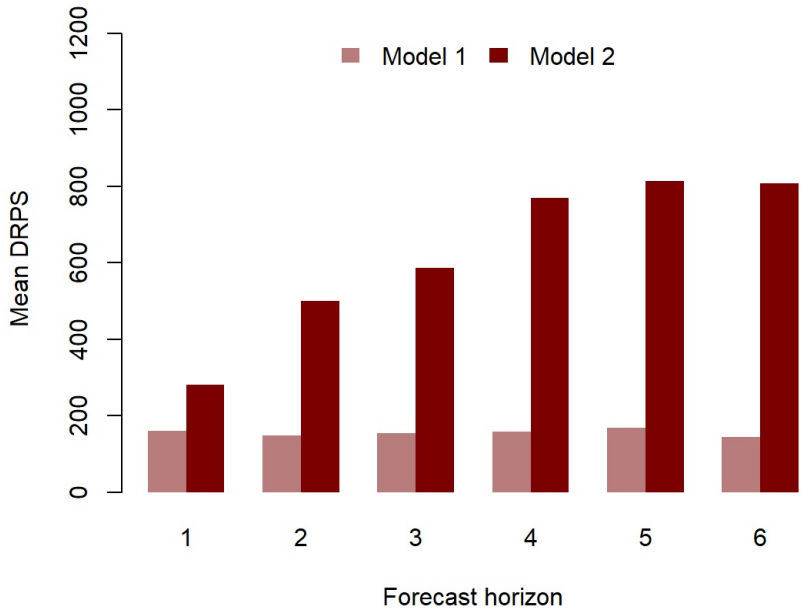
Now we will showcase how different dynamic models can be compared using rolling probabilistic forecast evaluation, which is especially useful if we don't already have out of sample observations for comparing forecasts. This function sets up a sequence of evaluation timepoints along a rolling window within 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 = mod3, model2 = mod4,
    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 1 143.4533 149.6063 155.8828 155.2076 158.943 168.5497
## Model 2 281.2945 521.7906 677.6101 626.2886 798.178 812.9988
##
## 90% interval coverages per model (closer to 0.9 is better)
## Model 1 0.9869231
## Model 2 0.9935897
```


Sum DRPS (lower is better)
0
500
1500
2500
3500
model 1
model 2

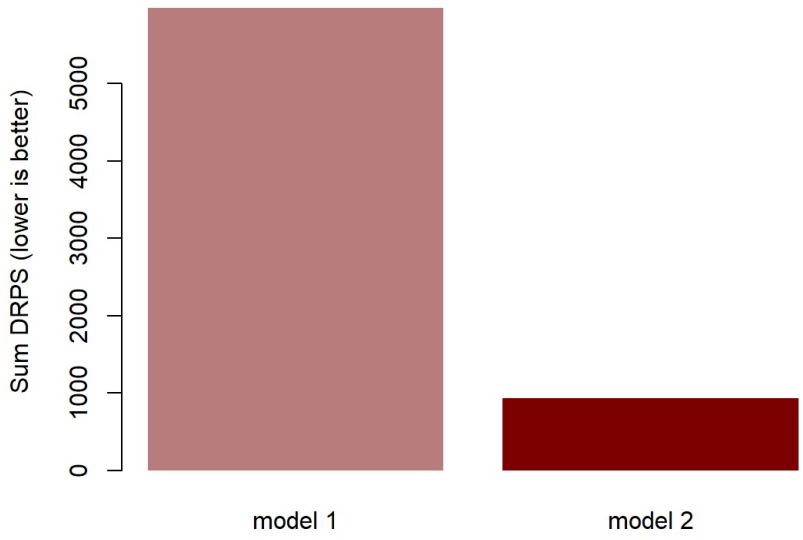

Sum DRPS per evaluation
200
300
400
500
600
700
800
model 1
model 2

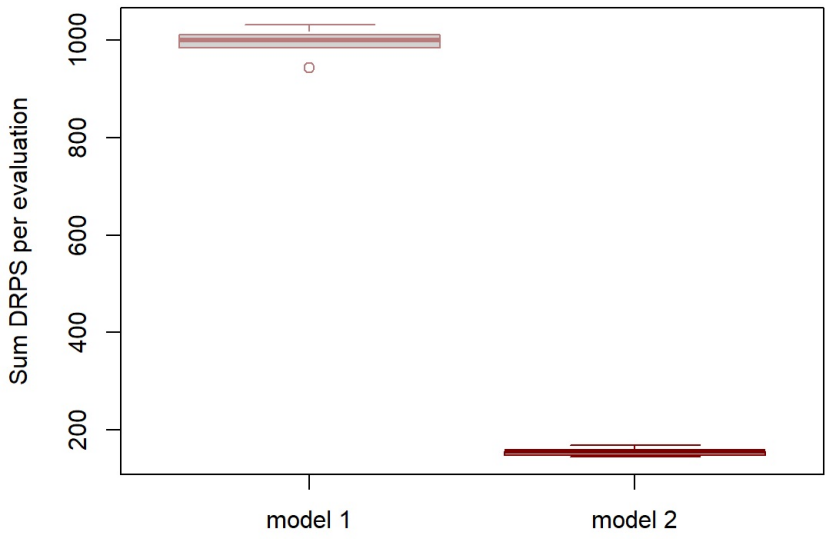


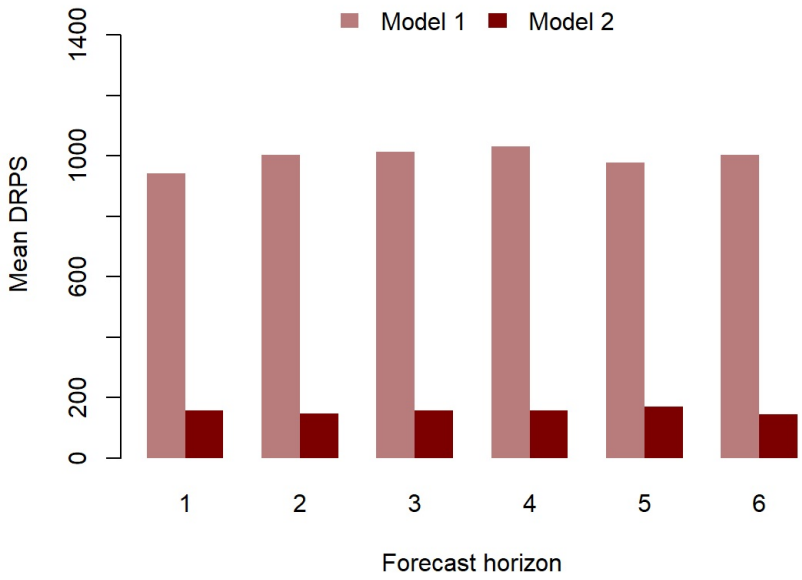
The series of plots generated by `compare_mvgams` clearly show that the first dynamic model generates better predictions. In each plot, DRPS for the forecast `horizon` is lower for the first model than for the second model. This kind of evaluation is often more appropriate for forecast models than complexity-penalising fit metrics such as `AIC` or `BIC` (https://www.sciencedirect.com/science/article/pii/S0169207020301096) However, comparing forecasts of the dynamic models against the two models with yearly smooth terms (`mod1` and `mod2`) using the rolling window approach is actually not recommended, as the yearly smooth models have already seen all the possible in-sample values of `year` and so should be able to predict incredibly well by interpolating through the range of the fitted smooth. By contrast, the dynamic component in our dynamic GAMs (models 3 and 4) produce true forecasts when running the rolling window approach. Nevertheless, when we compare some of these models (here `mod1` with the thin plate yearly smooth vs `mod3` with the AR3 trend process) as we did above for the random walk model, we still find that our dynamic GAM produces superior probabilistic forecasts

```
compare_mvgams(model1 = mod1, model2 = mod3,
    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 1 943.5635 984.9735 1004.1143 995.9697 1011.773 1031.1067
## Model 2 144.9278 148.3712  156.1984 154.8742  156.835  169.1112
##
## 90% interval coverages per model (closer to 0.9 is better)
## Model 1 1
## Model 2 0.9935897
```



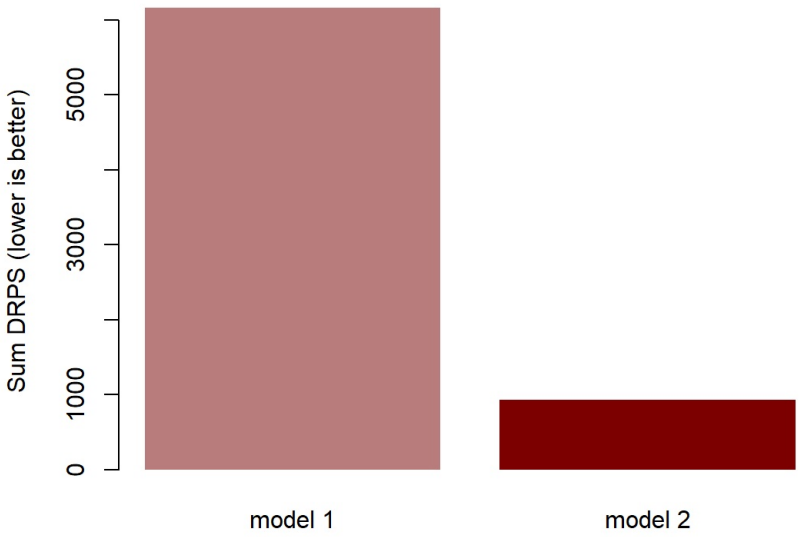


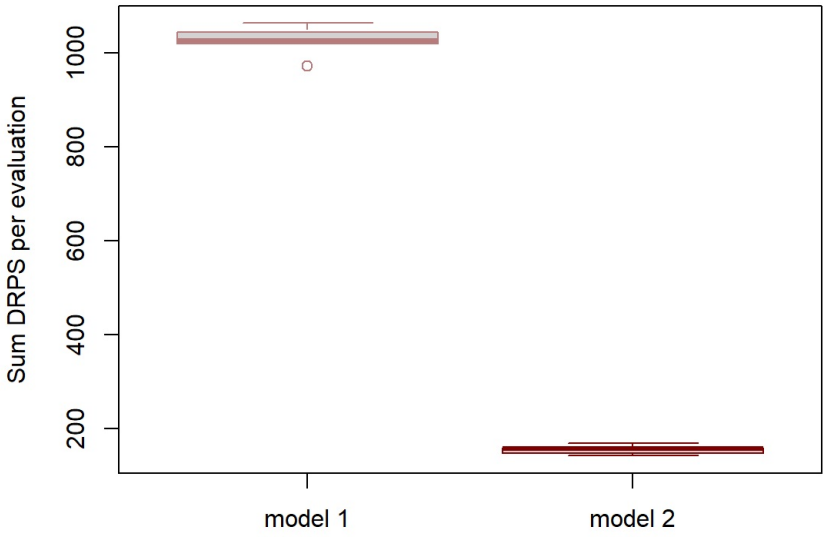


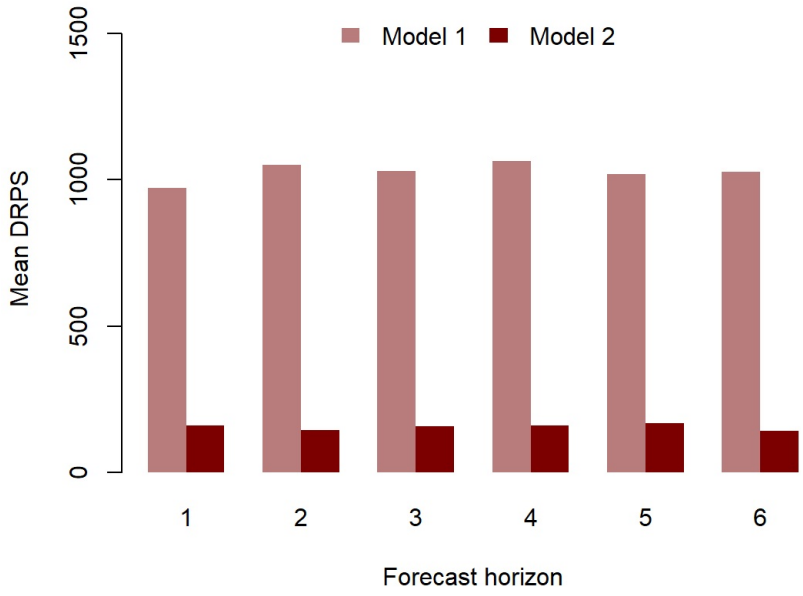
The same holds true when comparing against the `B` spline model (`mod2`)

```
compare_mvgams(model1 = mod2, model2 = mod3,
    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 1 973.5177 1020.7516 1027.904 1027.1553 1045.4345 1063.9614
## Model 2 141.7023  147.6101  157.419  154.7454  158.8552  168.2578
##
## 90% interval coverages per model (closer to 0.9 is better)
## Model 1 1
## Model 2 0.9802564
```


Sum DRPS (lower is better)
0
1000
3000
5000
model 1
model 2


Sum DRPS per evaluation
200
400
600
800
1000
model 1
model 2



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 `15000` 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 = mod3, n_particles = 15000,
    n_cores = 3, data_assim = fake_data$data_test)
```

```
## Saving particles to pfilter/particles.rda
##  ESS = 14999.56
```

Now we are ready to run the particle filter. This function will assimilate the next six 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 = fake_data$data_test[1:7,
    ], n_cores = 3, kernel_lambda = 1)
```

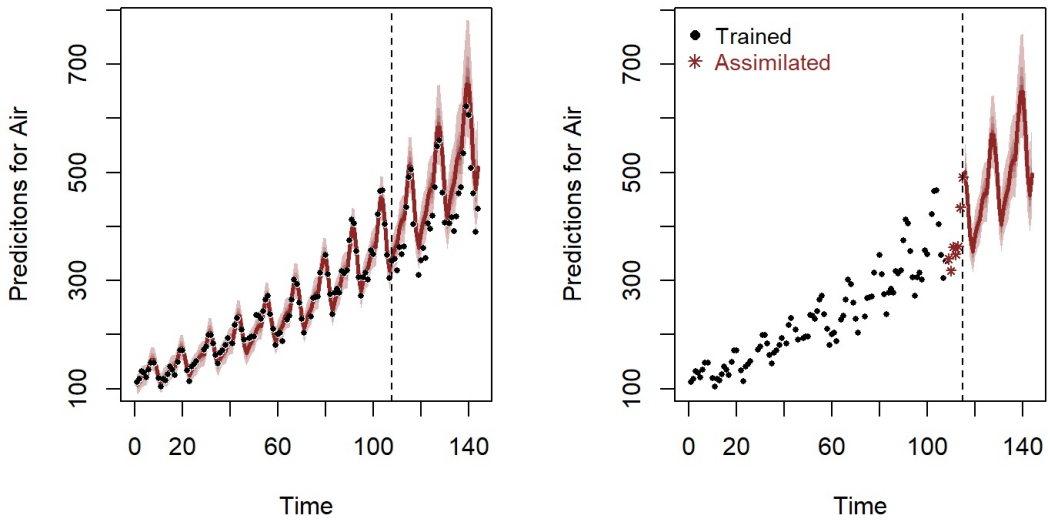
```
## Particles have already assimilated one or more observations. Skipping these
##
## Assimilating the next 6 observations
##
## Effective sample size is 14999.93 ...
##
## Smoothing particles ...
##
## Effective sample size is 14999.67 ...
##
## Smoothing particles ...
##
## Effective sample size is 14999.95 ...
##
## Smoothing particles ...
##
## Effective sample size is 14999.75 ...
##
## Smoothing particles ...
##
## Effective sample size is 14999.55 ...
##
## Smoothing particles ...
##
## Effective sample size is 14999.7 ...
##
## Smoothing particles ...
##
## Last assimilation time was 115
##
## Saving particles to pfilter/particles.rda
##  ESS = 15000
```

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 `year` and then by `season` 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.

```
fc <- pfilter_mvgam_fc(file_path = "pfilter",
    n_cores = 3, data_test = fake_data$data_test,
    ylim = c(min(fake_data$data_train$y,
        na.rm = T), max(fake_data$data_test$y,
        na.rm = T) * 1.25))
```

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

```
par(mfrow = c(1, 2))
plot_mvgam_fc(mod3, series = 1, data_test = fake_data$data_test,
    ylim = c(min(fake_data$data_train$y,
        na.rm = T), max(fake_data$data_test$y,
        na.rm = T) * 1.25))
fc$Air()
```



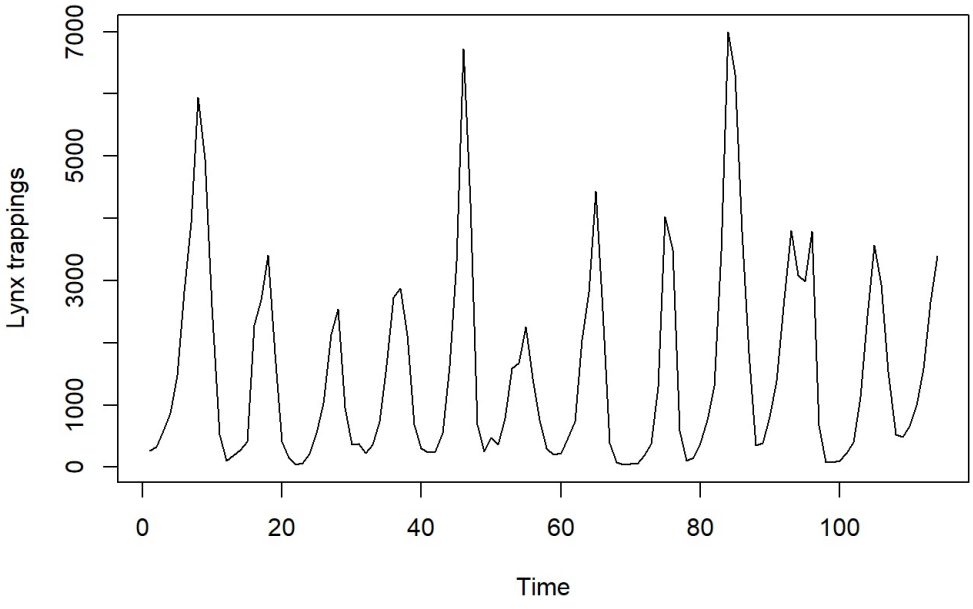
Here it is apparent that the distribution has shifted slightly in light of the `6` observations that have been assimilated, and that our confidence in the remaining forecast horizon has improved (tighter uncertainty intervals). This is an advantageous way of allowing a model to slowly adapt to new conditions while breaking free of restrictive assumptions about residual distributions. See some of the many particle filtering lectures by Nathaniel Osgood for more details (https://www.youtube.com/user/NathanielOsgood). Remove the particles from their stored directory when finished

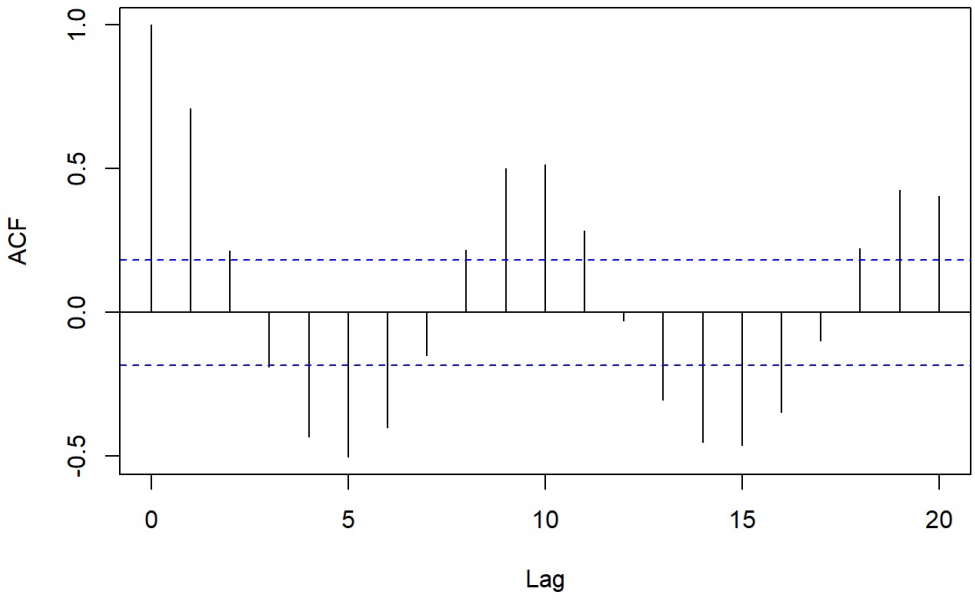
```
unlink("pfilter", recursive = T)
```

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 (https://noamross.github.io/mgcv-esa-2018/) 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

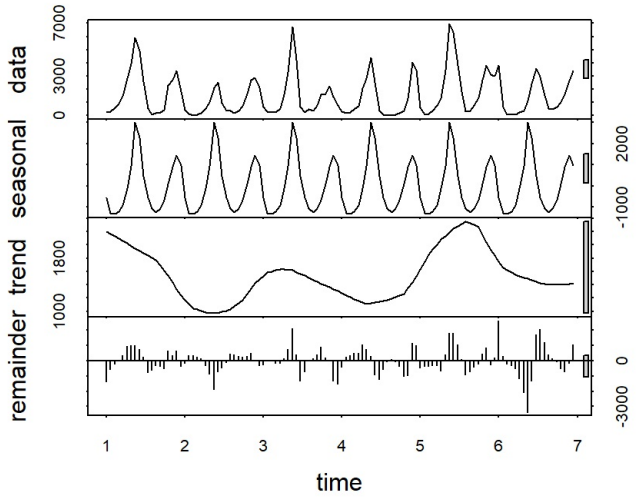
```
data(lynx)
lynx_full = data.frame(year = 1821:1934,
    population = as.numeric(lynx), time = 1:NROW(lynx))
plot(lynx_full$population, type = "l", ylab = "Lynx trappings",
    xlab = "Time")
acf(lynx_full$population, main = "")
```





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

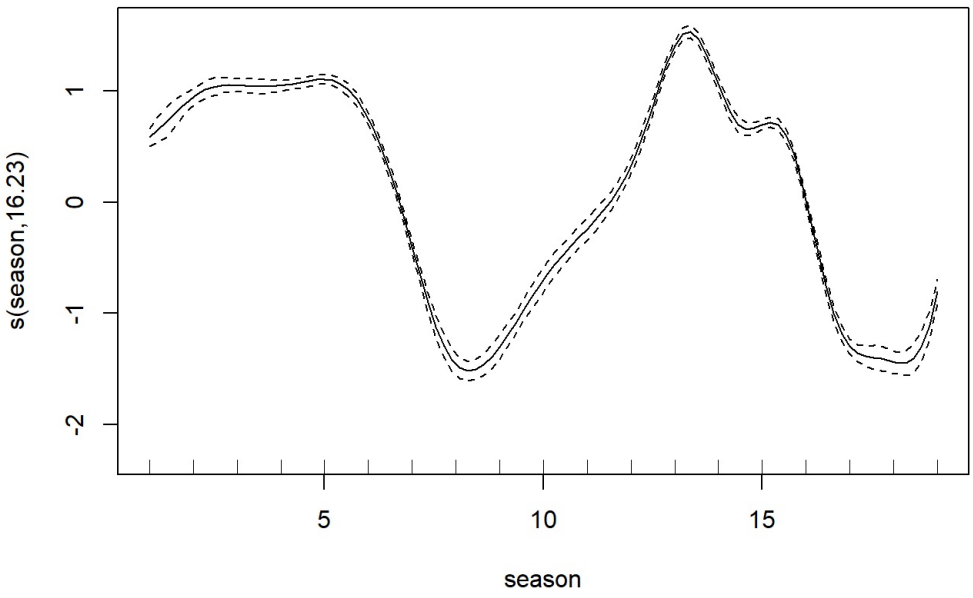
```
lynx_mgcv = gam(population ~ s(season, bs = "cc",
    k = 19) + s(year, bs = "bs", m = c(3,
    2, 1, 0)) + s(lag1, k = 5) + s(lag2,
    k = 5), knots = list(season = c(0.5,
    19.5), year = c(min(lynx_train$year -
    1), min(lynx_train$year), max(lynx_test$year),
    max(lynx_test$year) + 1)),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

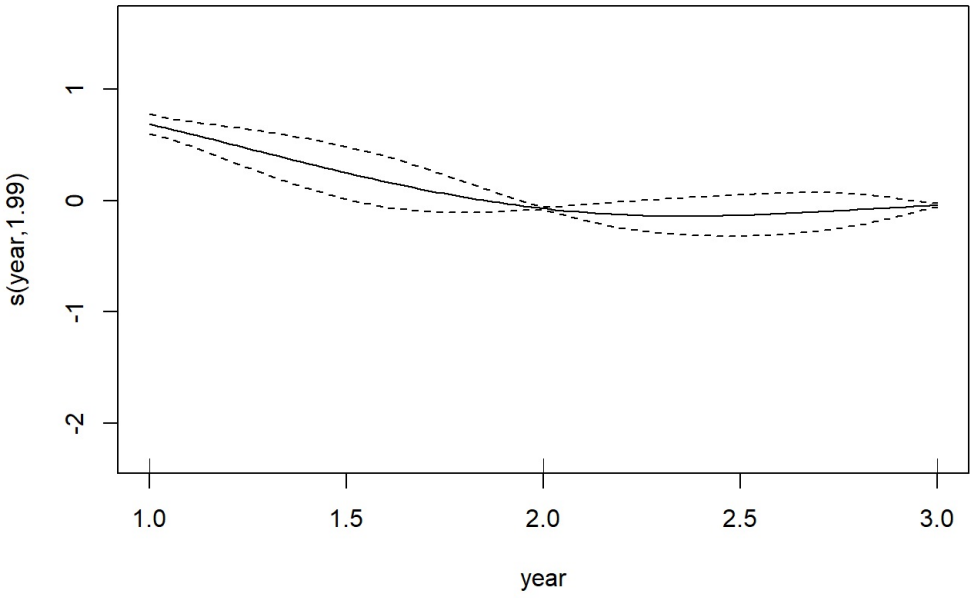
Inspect the model's summary and estimated smooth functions for the season, year and lag terms

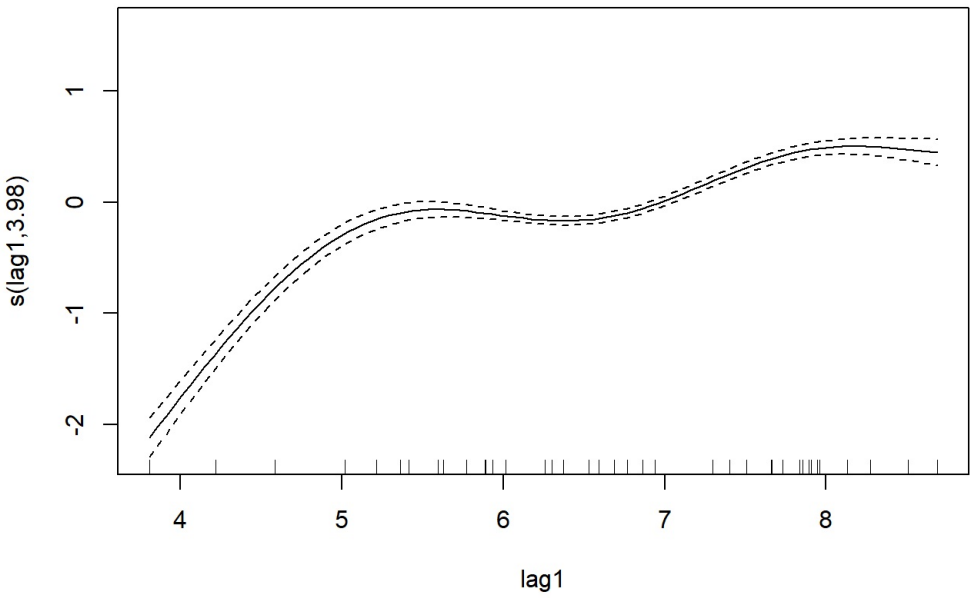
```
summary(lynx_mgcv)
```

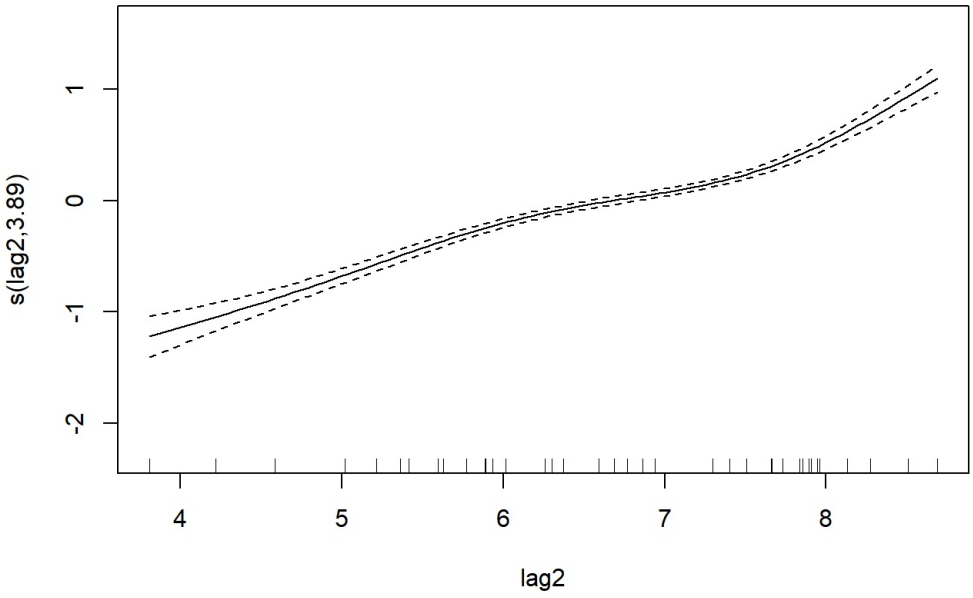
```
##
## Family: poisson
## Link function: log
##
## Formula:
## population ~ s(season, bs = "cc", k = 19) + s(year, bs = "bs",
##     m = c(3, 2, 1, 0)) + s(lag1, k = 5) + s(lag2, k = 5)
##
## Parametric coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 6.666631   0.007511   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)
plot(lynx_mgcv, select = 3)
plot(lynx_mgcv, select = 4)
```


s(season,16.23)
1
0
-1
-2
5
10
15
season


s(year,1.99)
1
0
-1
-2
1.0
1.5
2.0
2.5
3.0
year





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

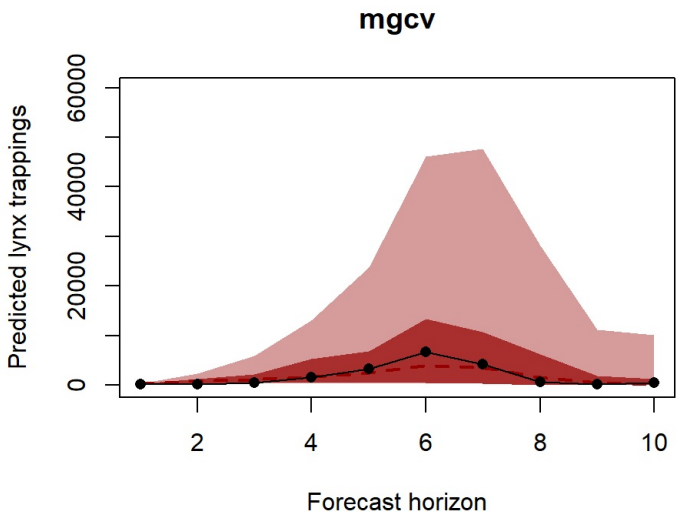
```
recurse_nonlin = function(model, lagged_vals,
    h) {
    # Initiate state vector
    states <- rep(NA, length = h + 2)
    # Last two values of the conditional
    # expectations begin the state vector
    states[1] <- as.numeric(exp(lagged_vals[2]))
    states[2] <- as.numeric(exp(lagged_vals[1]))
    # 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,
        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) hpd(x,
    0.95))
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) hpd(x,
    0.68))
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])
```

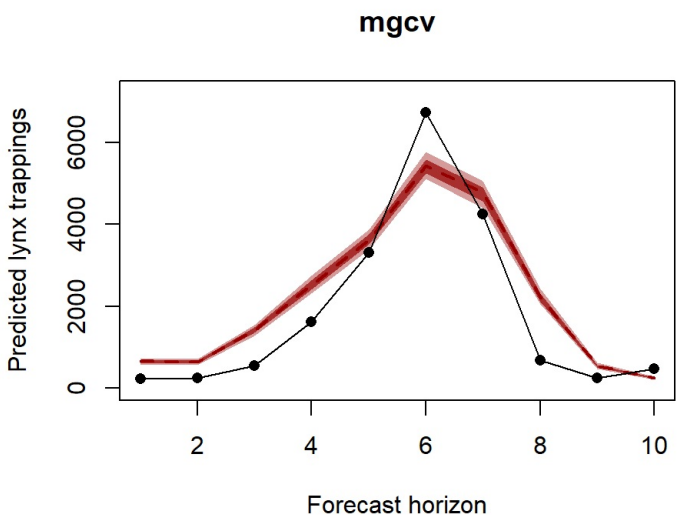


A decent forecast? The shape is certainly correct, but the 95% uncertainty intervals appear to be far too wide (i.e. our upper interval extends to up to ~8 times the maximum number of trappings that have ever been recorded up to this point). This is almost entirely due to the extrapolation behaviour 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

```
# 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 <- hpd(fc, interval_width)
    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)$bs
gam_sims <- matrix(NA, nrow = 1000, ncol = 10)
for (i in 1:1000) {
    gam_sims[i, ] <- recurse_nonlin(lynx_mgcv2,
        lagged_vals = c(lynx_test$lag1[1],
            lynx_test$lag2[1]), h = 10)
}
cred_ints <- apply(gam_sims, 2, function(x) hpd(x,
    0.95))
yupper <- max(cred_ints) * 1.25
plot(cred_ints[3, ] ~ seq(1:NCOL(cred_ints)),
    type = "l", col = rgb(1, 0, 0, alpha = 0),
    ylim = c(0, yupper), ylab = "Predicted lynx trappings",
    xlab = "Forecast horizon", main = "mgcv")
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
    c(cred_ints[1, ], rev(cred_ints[3, ])),
    col = rgb(150, 0, 0, max = 255, alpha = 100),
    border = NA)
cred_ints <- apply(gam_sims, 2, function(x) hpd(x,
    0.68))
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
    c(cred_ints[1, ], rev(cred_ints[3, ])),
    col = rgb(150, 0, 0, max = 255, alpha = 180),
    border = NA)
lines(cred_ints[2, ], col = rgb(150, 0, 0,
    max = 255), lwd = 2, lty = "dashed")
points(lynx_test$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)
```

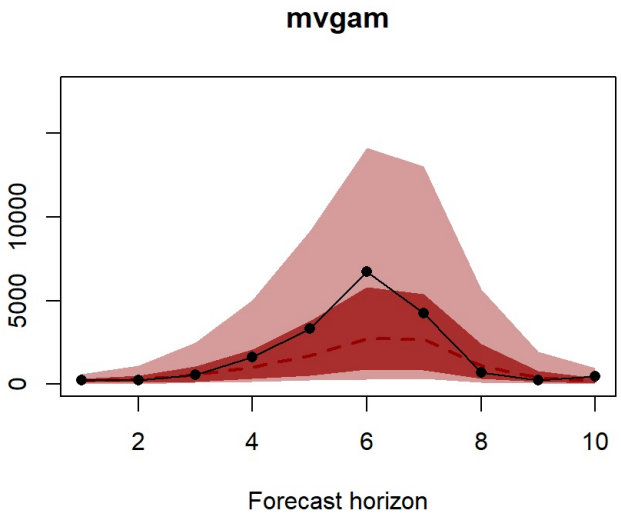
This forecast is highly overconfident, with very unrealistic uncertainty intervals due to the interpolation behaviours of the lag smooths. You can certainly keep trying different formulations (our experience is that the B spline variant above produces the best forecasts from any tested `mgcv` model, 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 <- mvjagam(data_train = lynx_train,
    data_test = lynx_test, formula = y ~
        s(season, bs = "cc", k = 19), knots = list(season = c(0.5,
        19.5)), family = "poisson", trend_model = "AR1",
    burnin = 5000, chains = 4)
```

```
## NOTE: Stopping adaptation
```

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

```
fits <- MCMCvis::MCMCchains(lynx_mvgam$jags_output,
    "ypred")
fits <- fits[, (NROW(lynx_mvgam$obs_data) +
    1):(NROW(lynx_mvgam$obs_data) + 10)]
cred_ints <- apply(fits, 2, function(x) hpd(x,
    0.95))
yupper <- max(cred_ints) * 1.25
plot(cred_ints[3, ] ~ seq(1:NCOL(cred_ints)),
    type = "l", col = rgb(1, 0, 0, alpha = 0),
    ylim = c(0, yupper), ylab = "", xlab = "Forecast horizon",
    main = "mvgam")
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
    c(cred_ints[1, ], rev(cred_ints[3, ])),
    col = rgb(150, 0, 0, max = 255, alpha = 100),
    border = NA)
cred_ints <- apply(fits, 2, function(x) hpd(x,
    0.68))
polygon(c(seq(1:(NCOL(cred_ints))), rev(seq(1:NCOL(cred_ints)))),
    c(cred_ints[1, ], rev(cred_ints[3, ])),
    col = rgb(150, 0, 0, max = 255, alpha = 180),
    border = NA)
lines(cred_ints[2, ], col = rgb(150, 0, 0,
    max = 255), lwd = 2, lty = "dashed")
points(lynx_test$population[1:10], pch = 16)
lines(lynx_test$population[1:10])
```



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

```
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)

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

```
## [1] 1533.365
```

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

```
## [1] 0.8
```

For the second `mgcv` model

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

```
## [1] 2042.308
```

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

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

And for the `mvgam` model

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

```
## [1] 767.8645
```

```
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 (https://robjhyndman.com/hyndsight/tscv/). 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)

```
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 per series:
```

```
## 40
```

```
##
```

```
## Status:
```

```
## Fitted using runjags::run.jags()
```

```
##
```

```
## GAM smooth term approximate significances:
```

```
##             edf Ref.df Chi.sq p-value
## s(season) 15.84  17.00  151.1  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
```

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

```
##                    2.5%         50%       97.5% Rhat n.eff
## (Intercept)   6.6005960  6.74556097  6.89139826 1.05    97
## s(season).1  -1.2234077 -0.59776729 -0.07619986 1.19    76
## s(season).2  -0.4522650  0.26490224  0.79716635 1.16    89
## s(season).3   0.2787372  0.96967471  1.59193618 1.11    49
## s(season).4   0.9669868  1.55296890  2.27880412 1.20    55
## s(season).5   1.3034314  1.77163694  2.64721795 1.39    45
## s(season).6   0.5977448  1.21213402  1.96314619 1.40    46
## s(season).7  -0.6265181  0.06597414  0.64860657 1.29    89
## s(season).8  -1.5498550 -0.84821492 -0.18264399 1.14   124
## s(season).9  -1.6380561 -0.96556724 -0.17437567 1.15   112
## s(season).10 -1.1530970 -0.54369241  0.35578965 1.16   118
## s(season).11 -0.3537029  0.24665510  1.17572816 1.16    71
## s(season).12  0.4197539  1.19151463  1.90235246 1.15    36
## s(season).13  0.6055738  1.39172001  2.06122704 1.16    30
## s(season).14  0.3768048  1.09337690  1.80938360 1.06    41
## s(season).15 -0.4654346  0.13115051  0.72273998 1.01    85
## s(season).16 -1.1414258 -0.62879942 -0.10247216 1.03   216
## s(season).17 -1.5138493 -1.00764125 -0.54377010 1.16   158
```

```
##
```

```
## GAM smoothing parameter (rho) estimates:
```

```
##               2.5%      50%    97.5% Rhat n.eff
## s(season) 3.598108 4.501352 5.240828 1.01   793
```

```
##
```

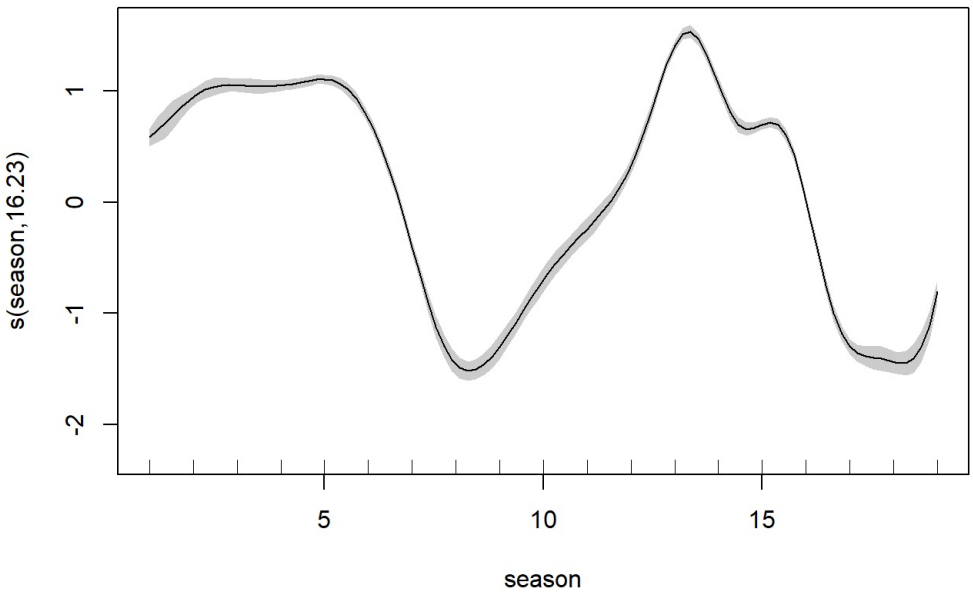
```
## Latent trend drift (phi) and AR parameter estimates:
```

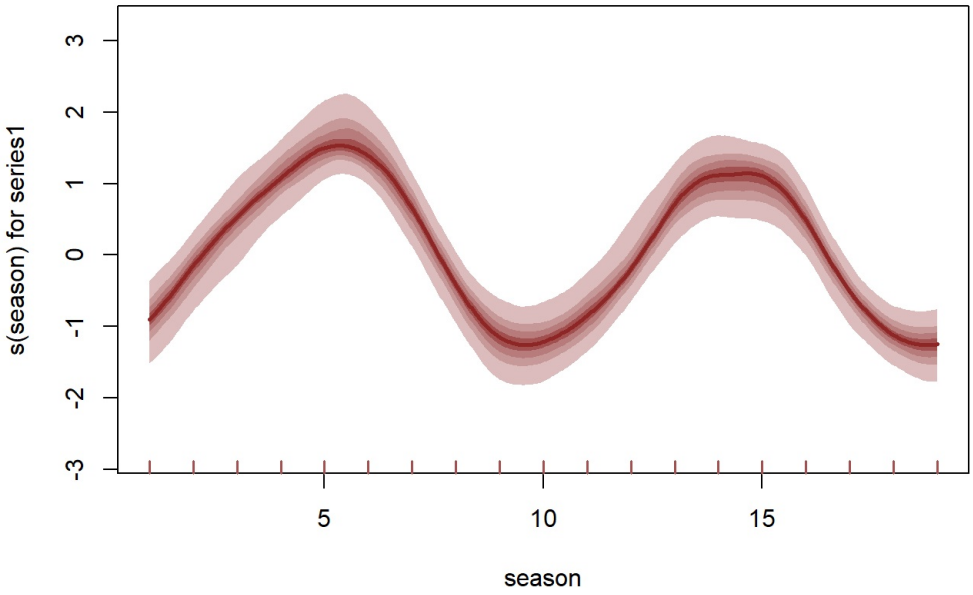
```
##          2.5%       50%     97.5% Rhat n.eff
## phi 0.0000000 0.0000000 0.0000000  NaN     0
## ar1 0.4568812 0.6899718 0.8938046 1.01  3137
## ar2 0.0000000 0.0000000 0.0000000  NaN     0
## ar3 0.0000000 0.0000000 0.0000000  NaN     0
```

```
##
```

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)

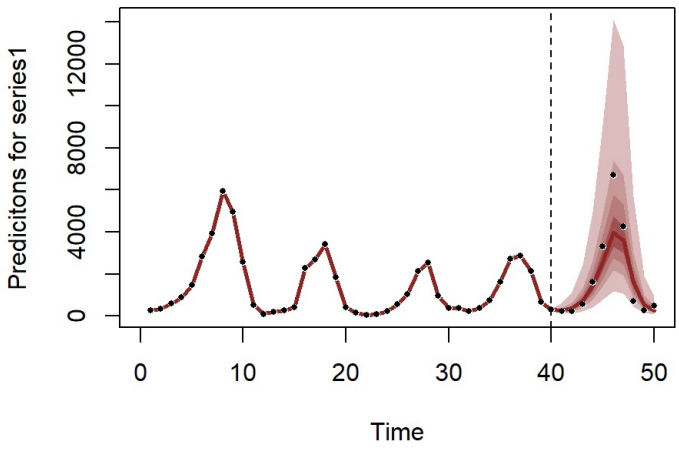
```
plot(lynx_mgcv, select = 1, shade = T)
plot_mvgam_smooth(lynx_mvgam, 1, "season")
```





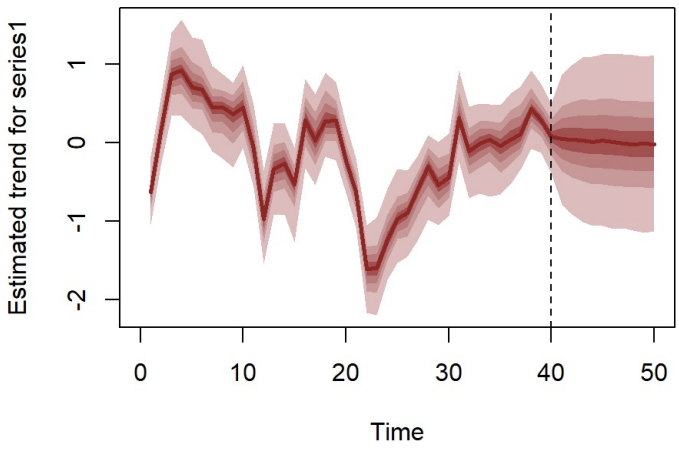
We can also view the mvgam's posterior predictions for the entire series (testing and training)

```
plot(lynx_mvgam, type = "forecast", data_test = lynx_test)
```



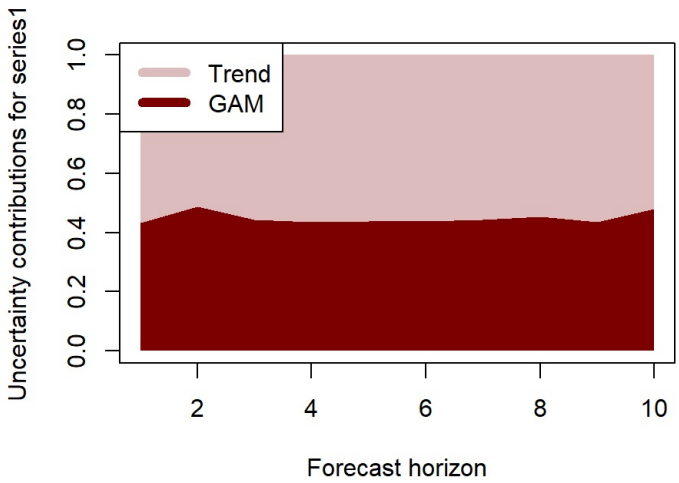
And the estimated trend

```
plot(lynx_mvgam, type = "trend", data_test = lynx_test)
```



A key aspect of ecological forecasting is to understand how different components of a model contribute to forecast uncertainty (https://esajournals.onlinelibrary.wiley.com/doi/full/10.1002/eap.1589). We can estimate contributions to forecast uncertainty for the GAM smooth functions and the latent trend using `mvgam`

```
plot(lynx_mvgam, type = "uncertainty", data_test = 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)
```