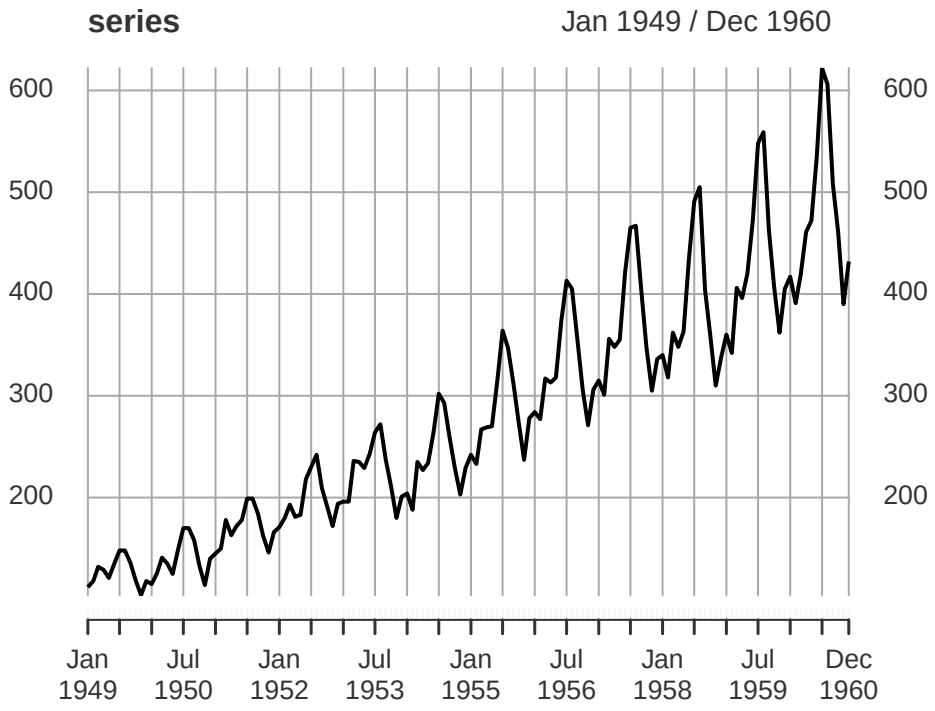
mvgam case study 1: model comparison and data assimilation

In this example we will examine some of the primary functions provided in `mvgam`, including methods to compare models based on rolling window forecasts and tools to assimilate new observations via a recursive particle filter. First load the `AirPassengers` data from the `forecast` package and convert to an `xts` object (which is easier to work with than a `ts` object)

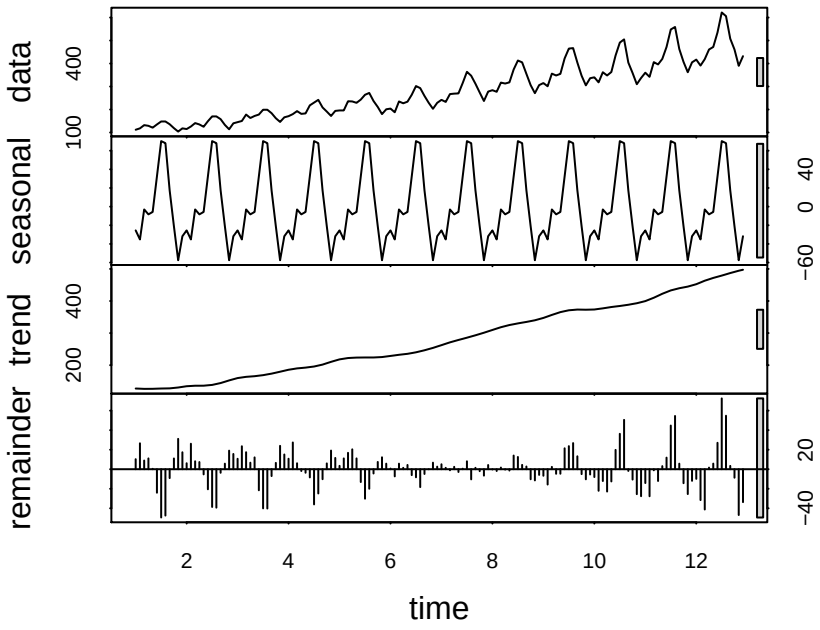
```
library(mvgam)
library(dplyr)
library(xts)
library(forecast)
data("AirPassengers")
series <- xts::as.xts(floor(AirPassengers))
colnames(series) <- c("Air")
```

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

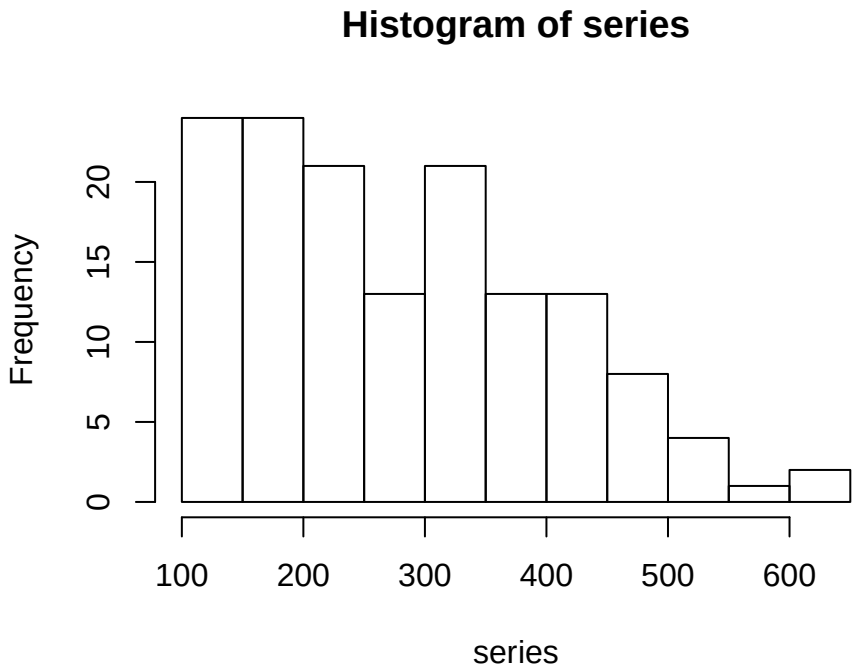
```
plot(series)
```




```
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

```
fake_data <- series_to_mvgam(series, freq = 12,
    train_prop = 0.75)
```

Examine the returned object

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

```
##     y season year       date series
## 1 112      1 1949 1949-01-01    Air
```

```
## 2 118      2 1949 1949-02-01    Air
## 3 132      3 1949 1949-03-01    Air
## 4 129      4 1949 1949-04-01    Air
## 5 121      5 1949 1949-05-01    Air
## 6 135      6 1949 1949-06-01    Air
```

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

```
##     y season year       date series
## 1 340      1 1958 1958-01-01    Air
## 2 318      2 1958 1958-02-01    Air
## 3 362      3 1958 1958-03-01    Air
## 4 348      4 1958 1958-04-01    Air
## 5 363      5 1958 1958-05-01    Air
## 6 435      6 1958 1958-06-01    Air
```

Fit a well-specified model in which the GAM component of the linear predictor captures the repeated seasonality (with a cyclic smooth) and the dynamic latent trend captures the residual process using AR parameters (up to order 3). We use the Negative Binomial family and a burnin length of `15000` iterations. 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

```
mod <- 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", n.burnin = 15000,
    auto_update = F)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 108
##    Unobserved stochastic nodes: 333
##    Total graph size: 3874
##
## Initializing model
```

Check the model summary to examine convergence for key parameters

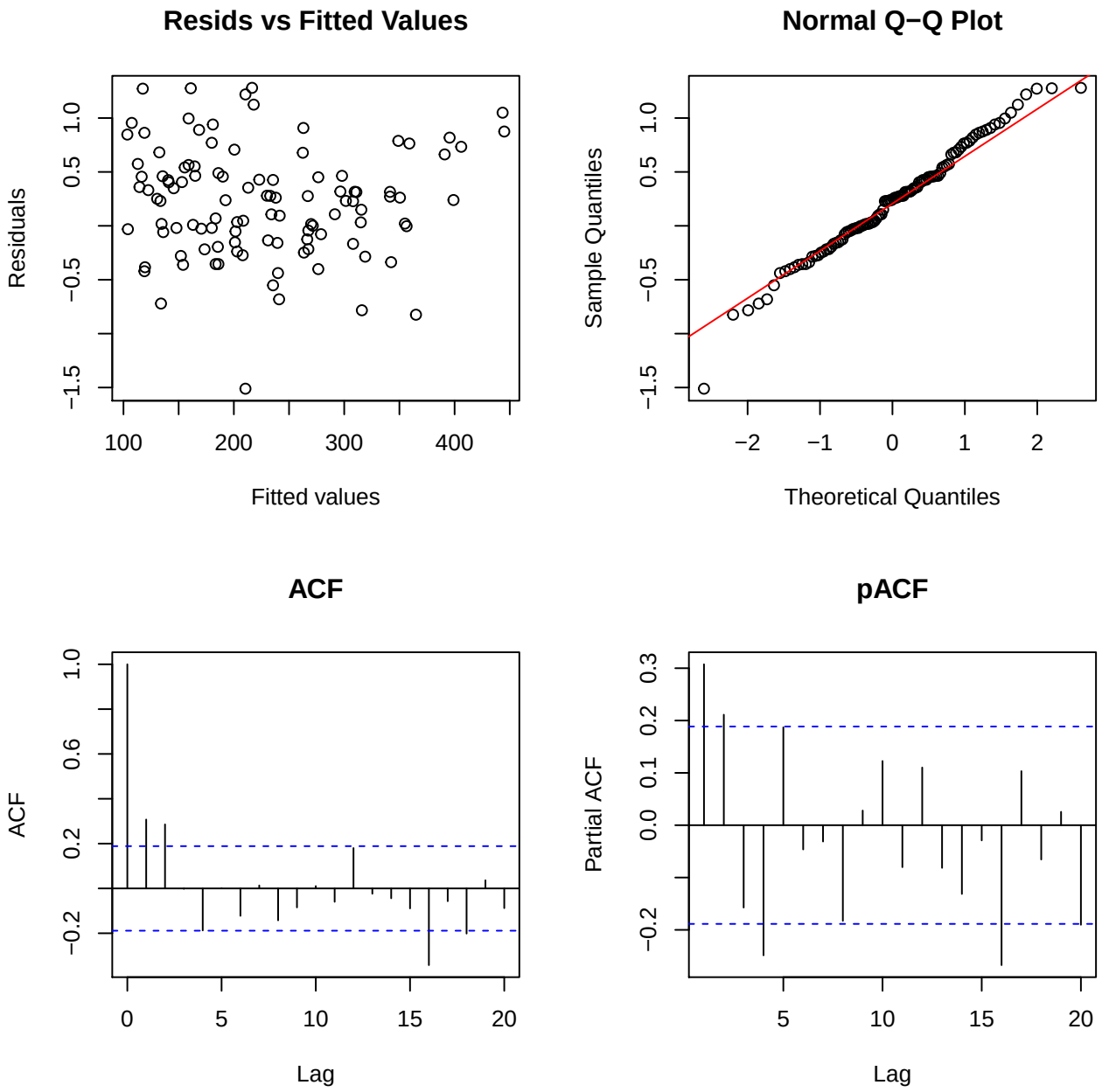
```
summary_mvgam(mod)
```

```
## GAM formula:

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

##

## Family:

## Negative Binomial

##

## N series:

## 1
```

```
##
## N observations per series:
## 108
##
## Dispersion parameter estimate:
##       2.5%      50%    97.5% Rhat n.eff
## r 1260.157 3104.171 6600.524    1  1403
##
## GAM smooth term approximate significances:
##              edf Ref.df Chi.sq p-value
## s(season)  7.805 10.000  369.1  <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)   4.702421559  4.760438781  4.820208513 1.02    10
## s(season).1  -0.154649385 -0.098505597 -0.042043880 1.08    82
## s(season).2  -0.097497704 -0.052492648 -0.001668062 1.16    99
## s(season).3   0.005090532  0.053080545  0.092481674 1.04   106
## s(season).4  -0.042919650 -0.002659492  0.039353014 1.01   119
## s(season).5   0.047414116  0.091956617  0.127977173 1.01   114
## s(season).6   0.196074083  0.230768340  0.270418243 1.01   121
## s(season).7   0.165258801  0.200348574  0.238435673 1.02   180
## s(season).8  -0.016598510  0.031815679  0.079552155 1.01   106
## s(season).9  -0.231996735 -0.185197552 -0.135289990 1.01    95
## s(season).10 -0.240989775 -0.179356325 -0.122735910 1.02    74
##
## GAM smoothing parameter (rho) estimates:
##               2.5%      50%    97.5% Rhat n.eff
## s(season) 4.596306 5.871509 6.811623 1.02   357
##
## Latent trend drift and AR parameter estimates:
##            2.5%        50%      97.5% Rhat n.eff
## phi  0.01313653 0.02452672 0.03671739    1   148
## ar1 -0.05218190 0.32187548 0.74946118    1   268
## ar2 -0.08070373 0.33106938 0.71051068    1   362
## ar3 -0.03515324 0.34743872 0.69496473    1   345
##
```

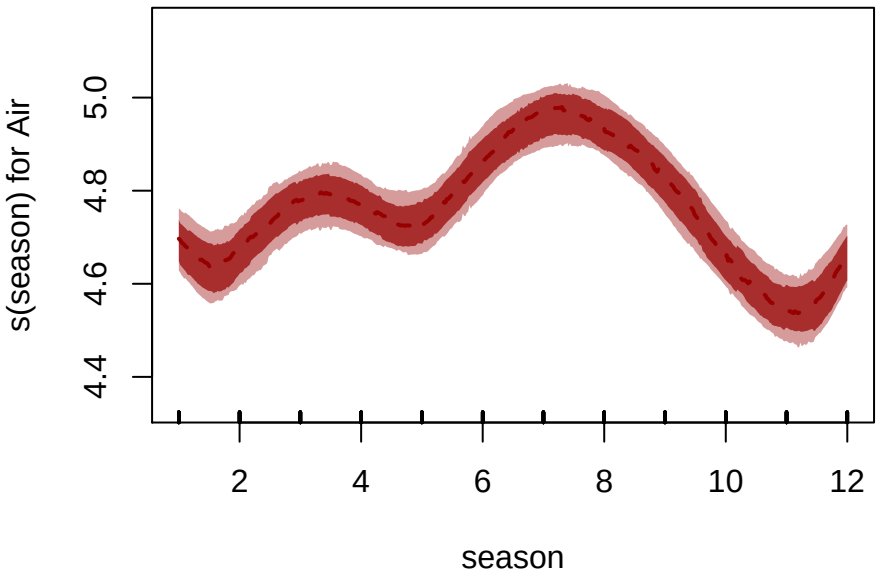
Plot diagnostics of posterior median Dunn-Smyth residuals to check for remaining autocorrelation and appropriateness of the Negative Binomial distribution

```
plot_mvgam_resids(mod, 1)
```



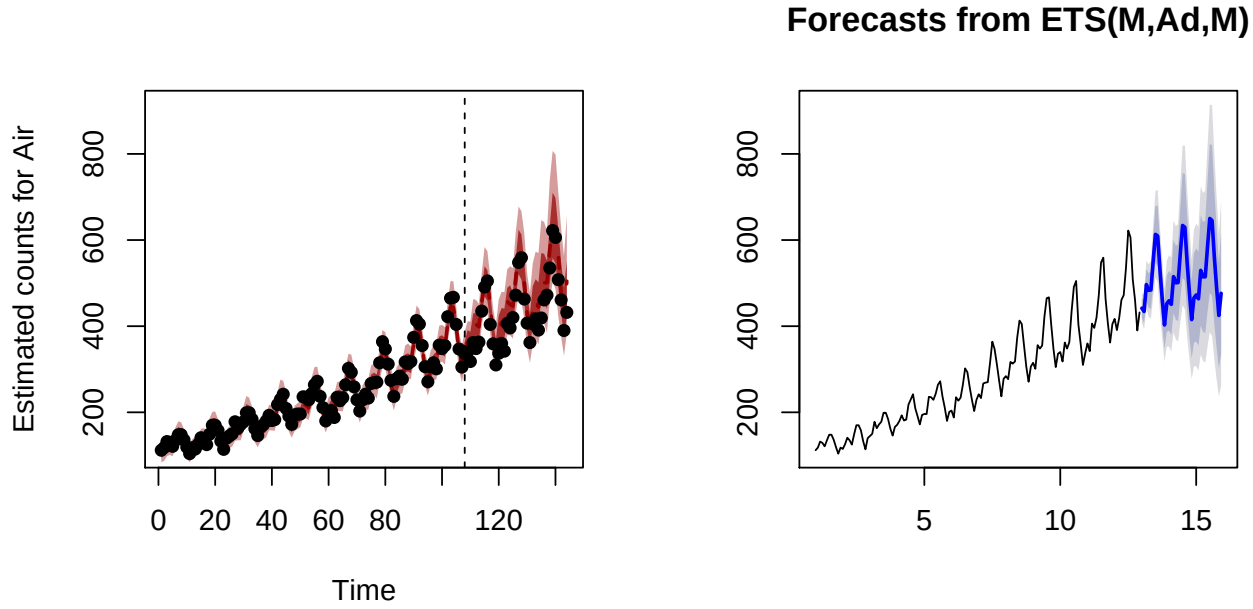
Plot the seasonal smooth term and its associated credible intervals

```
plot_mvgam_smooth(mod, 1, "season")
```



Compare the model's forecast distribution to one from an automatic exponential smoothing model (ETS) generated by the `forecast` package. This model is a useful benchmark for this series as the values are large enough that a Gaussian distribution is less inappropriate here. But note that for series with smaller counts, including zeros, as well as missing data and overdispersion, ETS models will not work. In fact, this lack of ability to fit some of the most popular automatic forecasting models to ecological time series was one of the primary motivations behind the design of the `mvgam` package

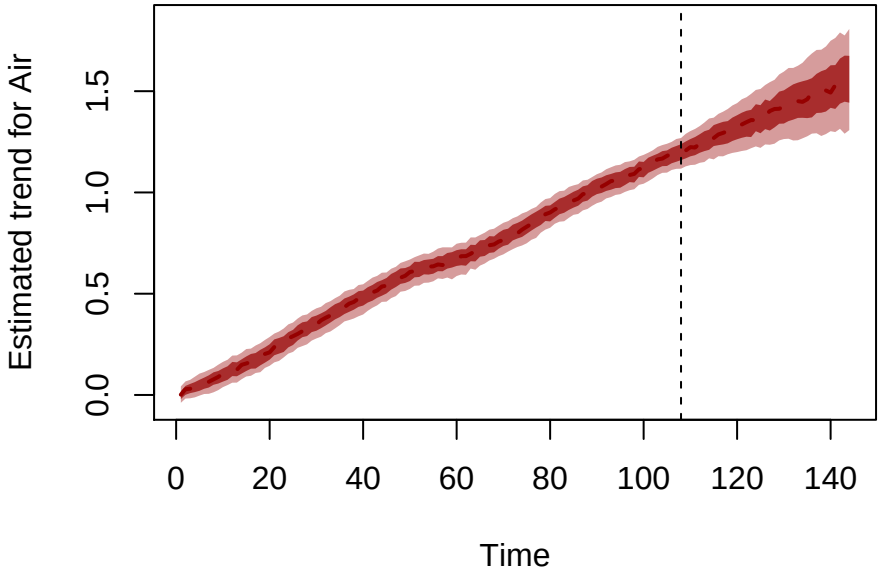
```
par(mfrow = c(1, 2))
ets_fc <- forecast(ets(series), h = NROW(fake_data$data_test))
plot_mvgam_fc(mod, ylim = c(min(ets_fc$x),
    max(ets_fc$upper)), data_test = fake_data$data_test)
plot(ets_fc)
```



The trends differ here because the ETS model uses a dampening parameter to ensure the trend flattens out into the future, which is a feature that has proven to improve forecasts over longer horizons. `mvgam` does not directly implement this feature (the AR terms will indirectly control the dampening of the trend), but this

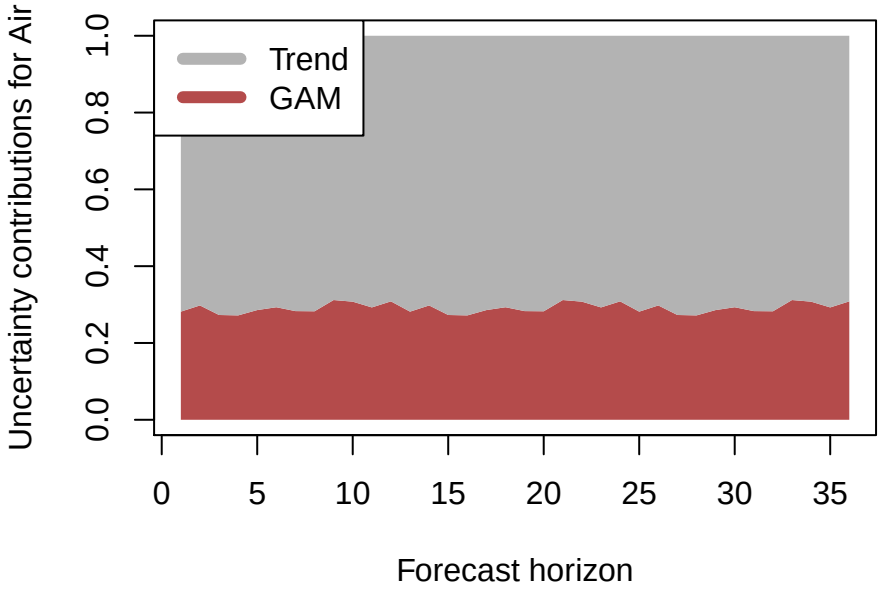
shouldn't be as much of an issue for near-term and medium-term predictions. Plot the estimated latent trend (which is on the log scale)

```
plot_mvgam_trend(mod, series = 1, data_test = fake_data$data_test)
```



Plot estimated contributions to forecast uncertainty

```
plot_mvgam_uncertainty(mod, series = 1, data_test = fake_data$data_test)
```



Benchmarking against simplistic "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`. First we fit a mis-specified 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))
mod2 <- 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", n.burnin = 1000,
    n.iter = 1000, thin = 1, auto_update = F)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 108
##    Unobserved stochastic nodes: 330
##    Total graph size: 2483
##
## Initializing model
```

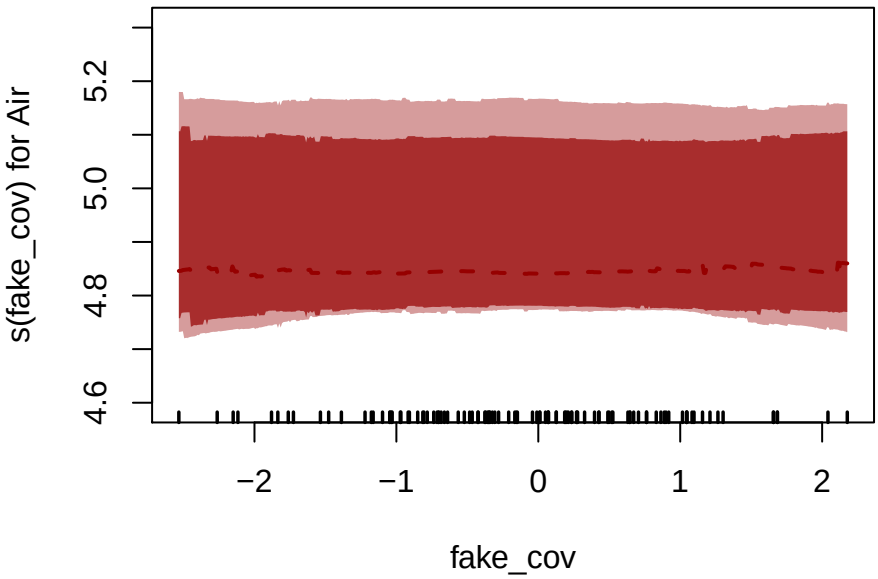
Look at the model's summary and key plots

```
summary_mvgam(mod2)
```

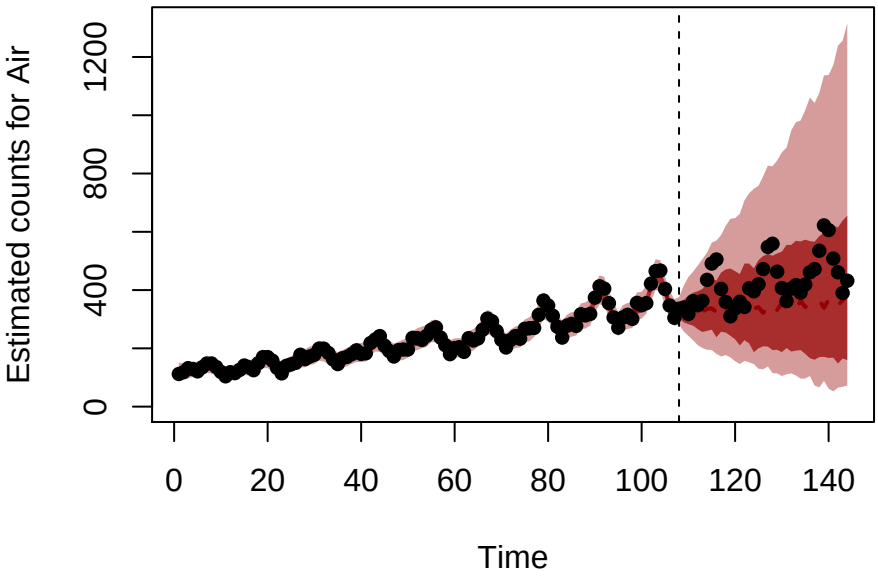
```
## GAM formula:
## y ~ s(fake_cov, k = 3)
##
## Family:
## Poisson
##
## N series:
## 1
##
## N observations per series:
## 108
##
## GAM smooth term approximate significances:
##               edf Ref.df Chi.sq p-value
## s(fake_cov) 1.659  1.883  0.005   0.997
##
## GAM coefficient (beta) estimates:
##                      2.5%           50%      97.5% Rhat n.eff
## (Intercept)    4.77136035  4.9251567724 5.16243141 5.38    12
## s(fake_cov).1 -0.04335192  0.0082371320 0.06762135 1.00   578
## s(fake_cov).2 -0.01941879 -0.0009999629 0.01909711 1.01   306
##
## GAM smoothing parameter (rho) estimates:
```

```
##                    2.5%      50%    97.5% Rhat n.eff
## s(fake_cov)  -1.654119 3.565299 5.750959 1.02   250
## s(fake_cov)2 -1.480098 4.211763 6.480816 1.04   302

##

## Latent trend drift and AR parameter estimates:

##              2.5%         50%      97.5% Rhat n.eff
## phi -0.008335773 0.008979988 0.02703138    1  1985
## ar1  1.000000000 1.000000000 1.00000000  NaN     0
## ar2  0.000000000 0.000000000 0.00000000  NaN     0
## ar3  0.000000000 0.000000000 0.00000000  NaN     0

##
```

```
plot_mvgam_smooth(mod2, 1, "fake_cov")
```



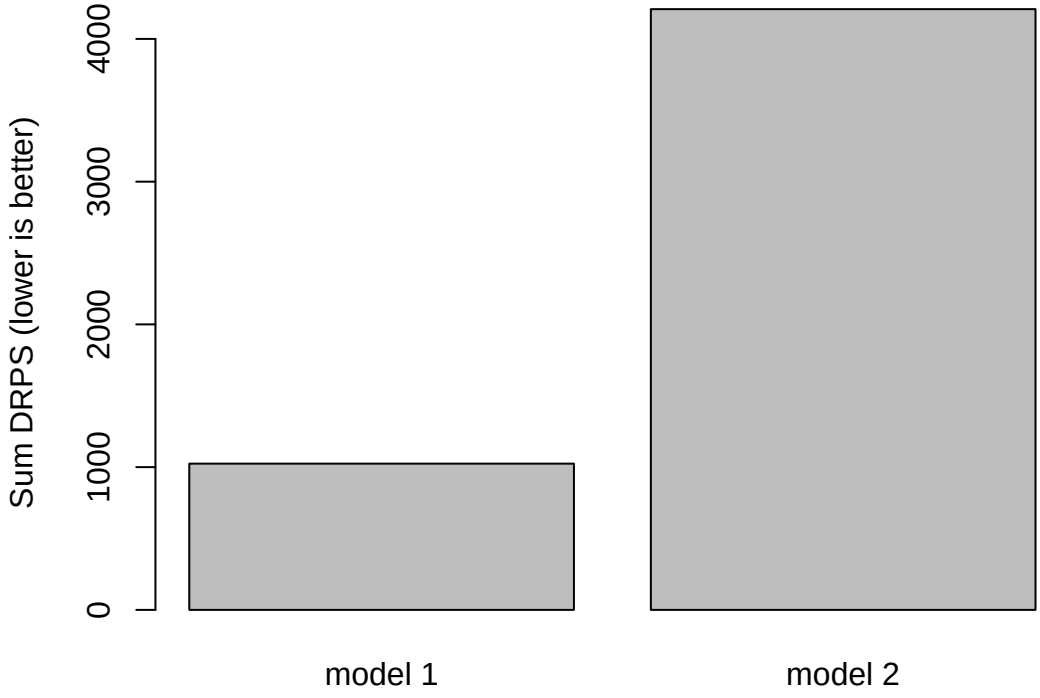

```
plot_mvgam_fc(mod2, 1, data_test = fake_data$data_test)
```

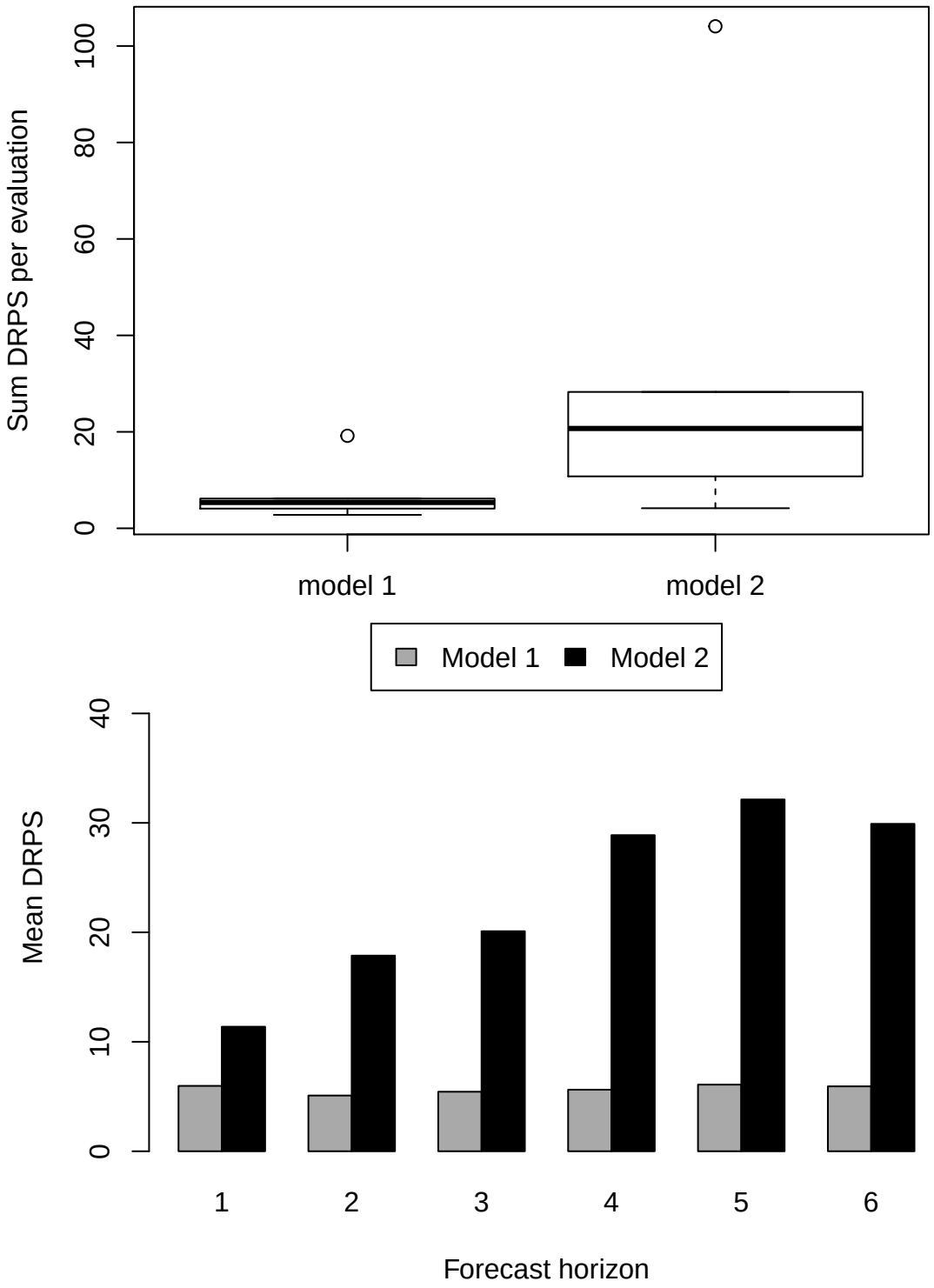


Now we compare the models using rolling probabilistic forecast evaluation. This function sets up a sequence of evaluation timepoints along a rolling window within the training data to evaluate 'out-of-sample' forecasts. The trends are rolled forward a total of `fc_horizon` timesteps according to their 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(mod, mod2, 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 2.786398  4.087879  5.026818  5.691584  6.166977  19.18391
## Model 2 4.162331 10.769249 18.019003 23.380265 28.280383 104.08631
##
## 90% interval coverages per model (closer to 0.9 is better)
## Model 1 1
## Model 2 0.95
```





The series of plots generated by `compare_mvgams` clearly show that the first model generates better predictions. In each plot, DRPS for the out of sample `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`. Obviously the autocorrelation in the data indicates that we probably need the dynamic trend model, but we can also explore how our inferences might change if we ignored this autocorrelation process. Here we fit our original model again but set `trend_model = "none"` and use a 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. 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

```
mod_notrend <- 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(3, 2,
                1, 0)), 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",
    n.burnin = 15000, auto_update = F)
```

```
## Warning in smooth.construct.bs.smooth.spec(object, dk$data, dk$knots): there is
## *no* information about some basis coefficients
```

```
## Warning in smooth.construct.bs.smooth.spec(object, dk$data, dk$knots): basis
## dimension is larger than number of unique covariates
```

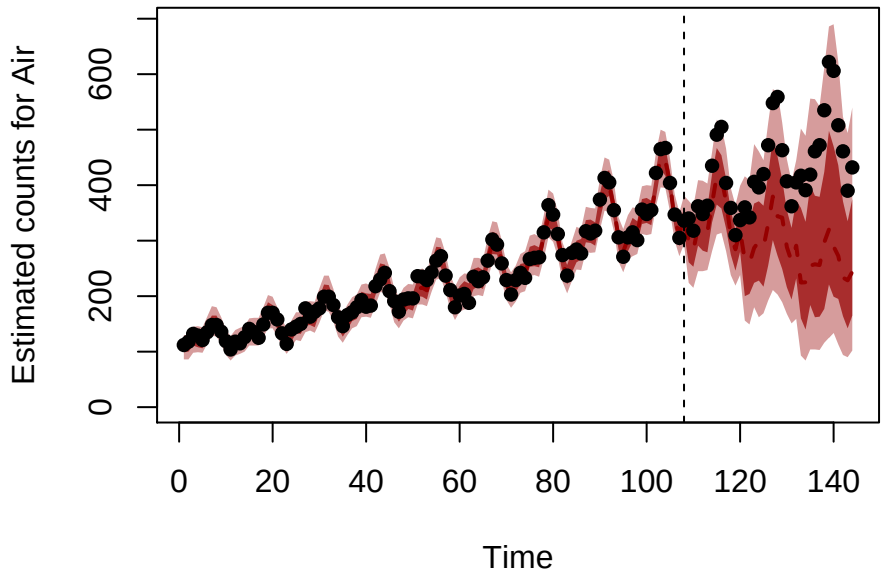
```
## Warning in smooth.construct.bs.smooth.spec(object, dk$data, dk$knots): there is
## *no* information about some basis coefficients
```

```
## Warning in smooth.construct.bs.smooth.spec(object, dk$data, dk$knots): basis
## dimension is larger than number of unique covariates
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 108
##    Unobserved stochastic nodes: 333
##    Total graph size: 5149
##
## Initializing model
```

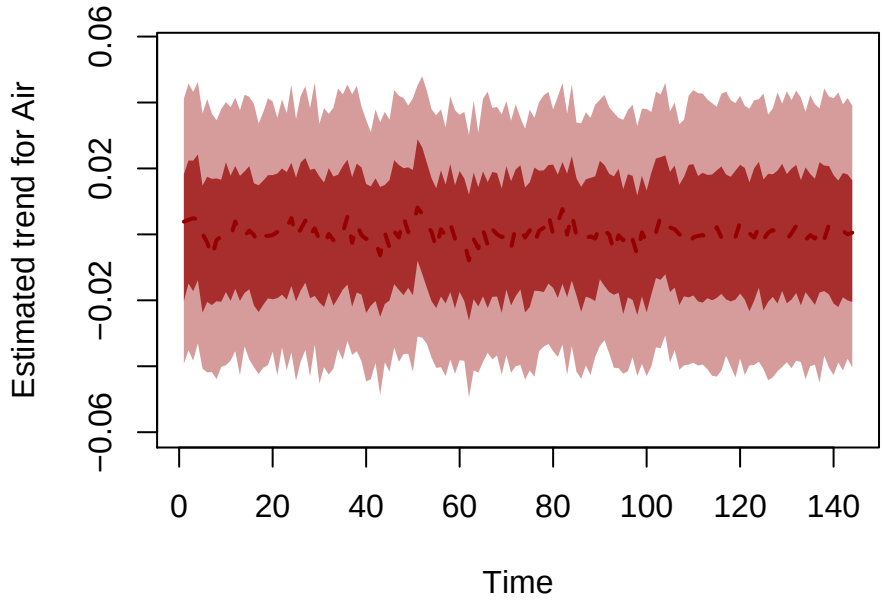
Have a look at the forecast. The yearly trend is being extrapolated into the future, which controls most of the shape and uncertainty in the forecast

```
plot_mvgam_fc(mod_notrend, 1, data_test = fake_data$data_test)
```



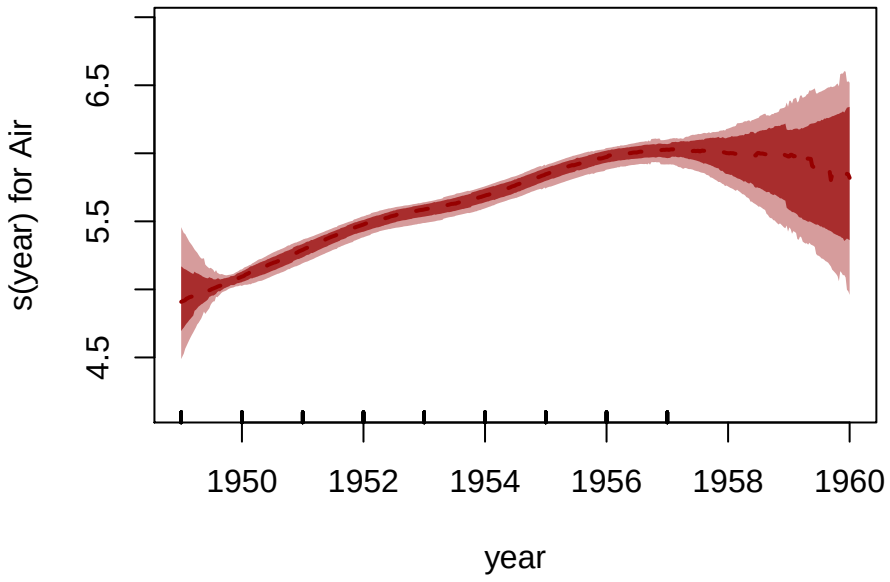
The "trend" model in this case is just the estimated static residual process

```
plot_mvgam_trend(mod_notrend, 1)
```



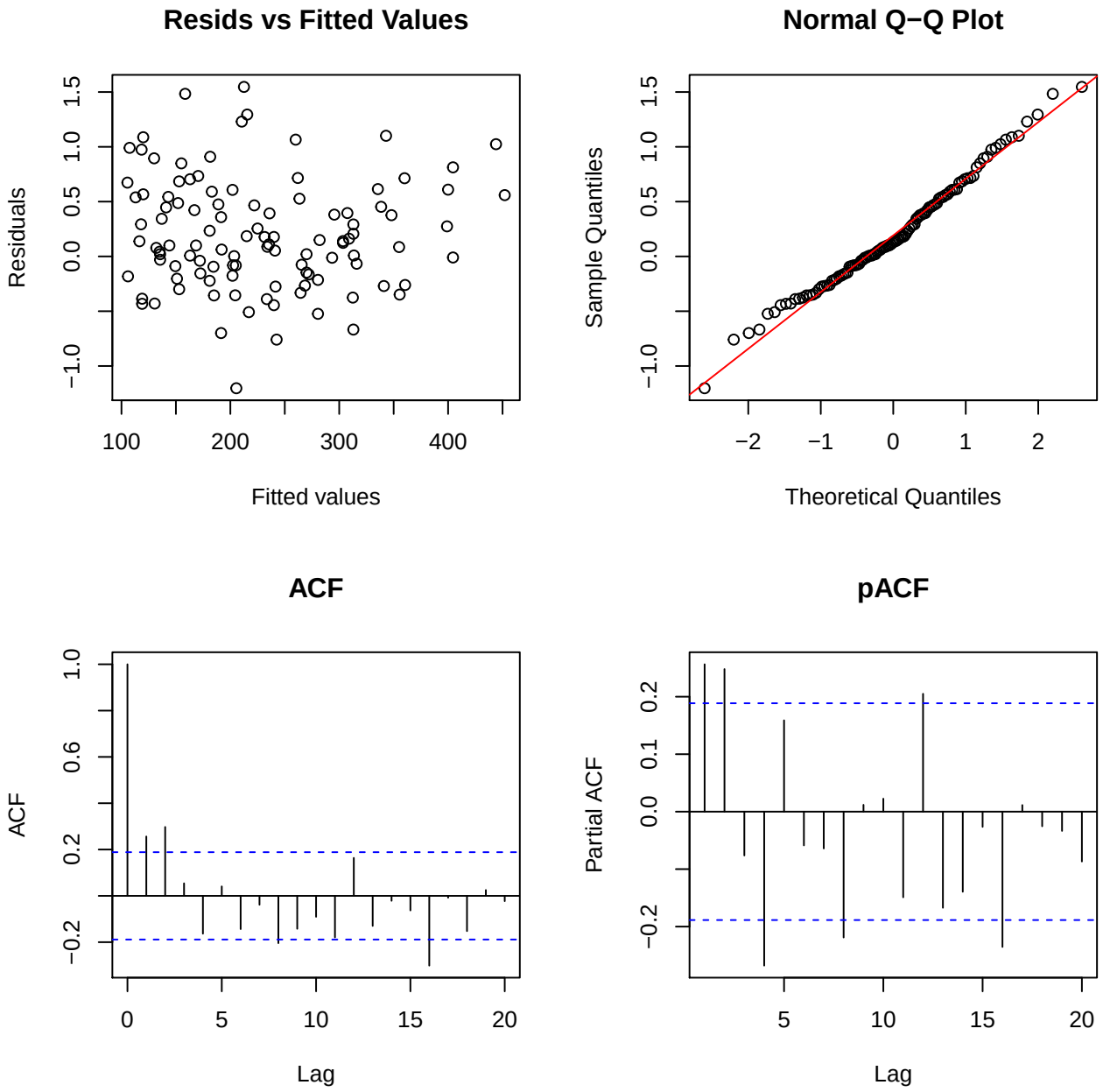
Here is the smooth for `year` over the training and testing values, which highlights how the B spline is starting to extrapolate

```
plot_mvgam_smooth(mod_notrend, 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)))
```



And here are the residual diagnostics

```
plot_mvgam_resids(mod_notrend, 1)
```

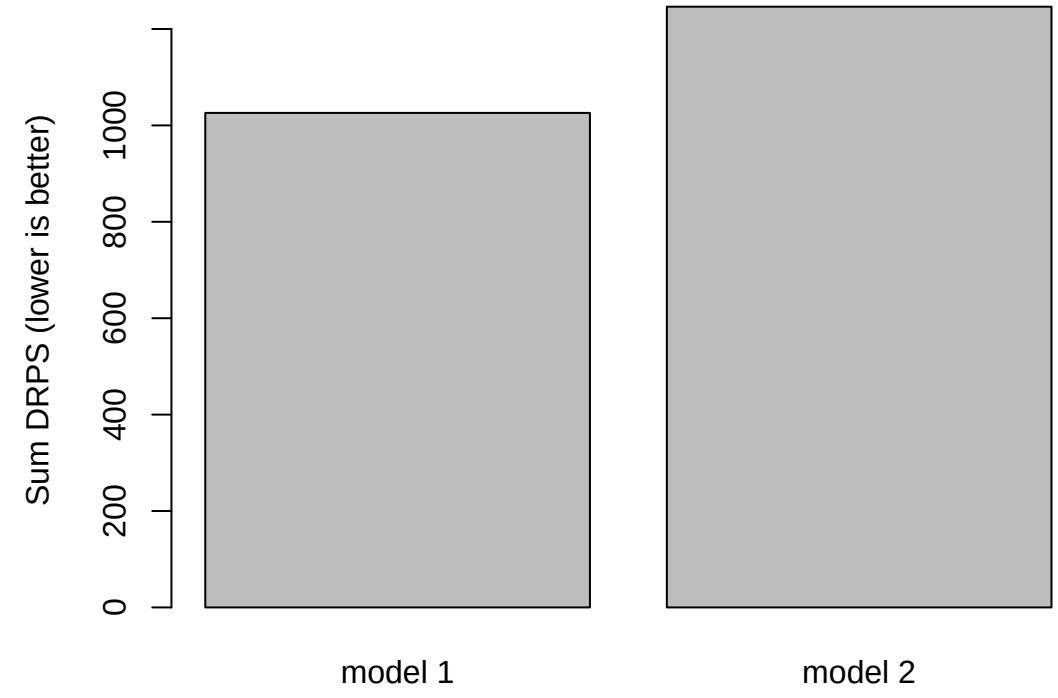


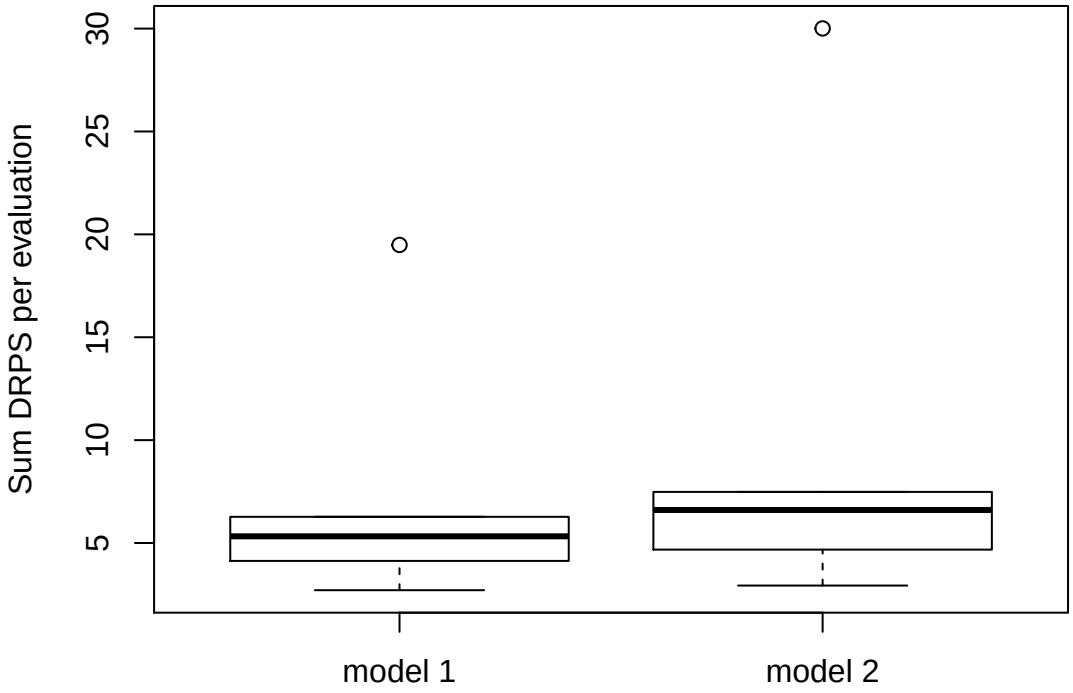
Comparing forecasts of these models using the rolling window approach is actually not recommended, as the second model has already seen all the possible in-sample values of `year` and so should be able to predict incredibly well by interpolating through the range of the fitted smooth. By contrast, the dynamic component in the first model produces true forecasts when running the rolling window approach. Nevertheless, when we compare the two models as we did above for the random walk model, we still find that our dynamic GAM produces better probabilistic forecasts

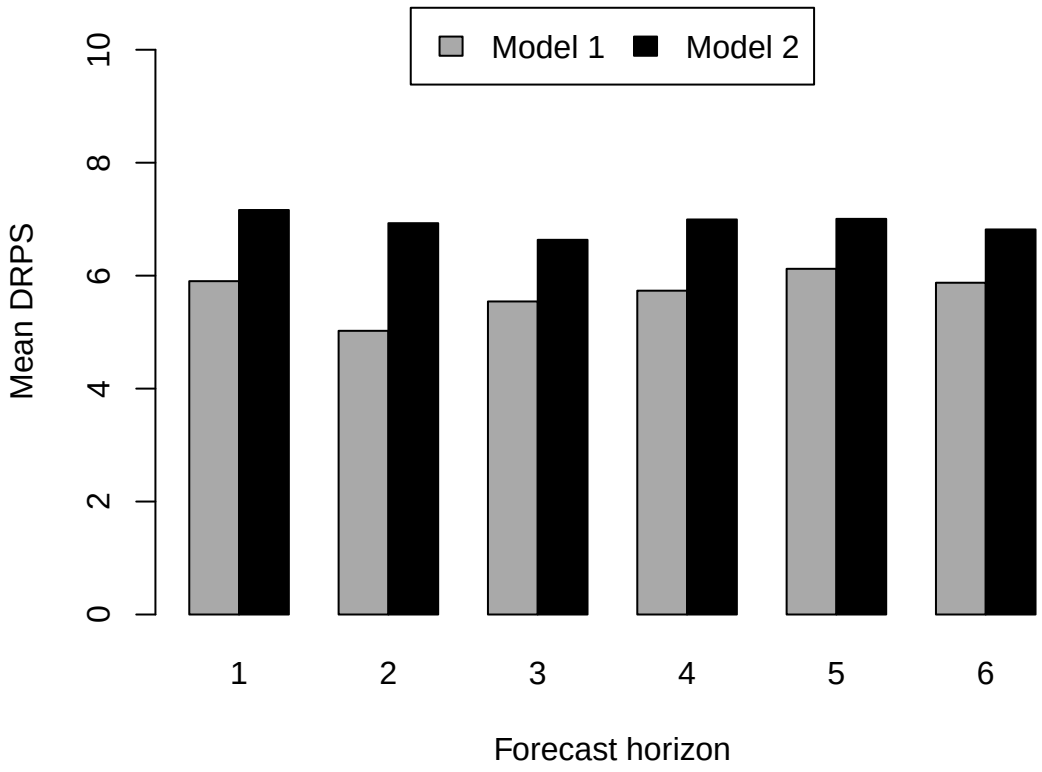
```
compare_mvgams(mod, mod_notrend, fc_horizon = 6,
    n_evaluations = 30, n_cores = 3)
```

```
## DRPS summaries per model (lower is better)
##             Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## Model 1 2.706364 4.129803 4.950271 5.699609 6.273782 19.48584
## Model 2 2.932377 4.678802 6.286641 6.923886 7.483808 30.00764
##
```

```
## 90% interval coverages per model (closer to 0.9 is better)
## Model 1 1
## Model 2 0.9944444
```







A better way of evaluating these models would be to compare their forecasts for the true out of sample period (using observations represented in `fake_data$data_test`). But we can see from comparing their forecast plots that the dynamic model clearly performs better. It is likely that, given the rather linear trend in the series, using a second derivative penalty to allow the spline to extrapolate linearly would lead to improved forecasts. But the uncertainties would still be poorly calibrated compared tot he dynamic `mvgam` model. Now we proceed to 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 latent trend component). The next observation in `data_assim` is assimilated and particles are weighted by how well their proposal (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 `10000` 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 = mod, n_particles = 10000,
    n_cores = 2, data_assim = fake_data$data_test)
```

```
## Saving particles to pfilter/particles.rda
##  ESS = 57.8567
```

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 = 2, kernel_lambda = 1)
```

```
## Particles have already assimilated one or more observations. Skipping these
##
## Assimilating the next 6 observations
##
## Effective sample size is 28.66251 ...
##
## Smoothing particles ...
##
## Effective sample size is 416.0329 ...
##
## Smoothing particles ...
##
## Effective sample size is 1147.062 ...
##
## Smoothing particles ...
##
## Effective sample size is 5509.35 ...
##
## Smoothing particles ...
##
## Effective sample size is 8336.181 ...
##
## Smoothing particles ...
##
## Effective sample size is 8936.216 ...
##
## Smoothing particles ...
##
## Last assimilation time was 7 1958
##
## Saving particles to pfilter/particles.rda
##  ESS = 8936.216
```

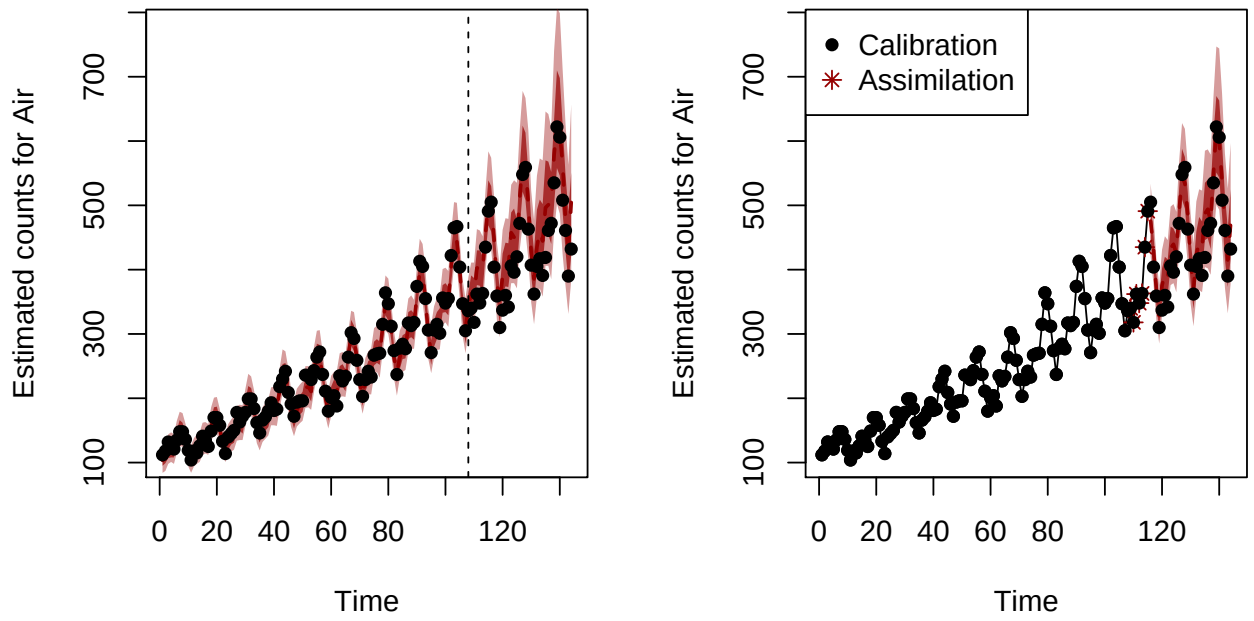
Once assimilation is complete, generate the updated forecast from the particles using the covariate information in remaining `data_test` observations. This function is designed to hopefully make it simpler to assimilate observations, as all that needs to be provided once the particles are initiated as 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 = 2, data_test = fake_data$data_test,
    ylim = c(min(fake_data$data_train$y),
        max(fake_data$data_test$y) * 1.25))
```

Compare the updated forecast to the original forecast to see how it has changed in light of the most recent observations

```
par(mfrow = c(1, 2))
plot_mvgam_fc(mod, series = 1, data_test = fake_data$data_test,
    ylim = c(min(fake_data$data_train$y),
```

```
        max(fake_data$data_test$y) * 1.25))
fc$Air()
points(c(fake_data$data_train$y, fake_data$data_test$y),
    pch = 16)
```



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. Remove the particles from their stored directory when finished

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