Class 3: Multivariate time series models, Gaussian processes, and co-integration

Andrew Parnell
andrew.parnell@ucd.ie

Learning outcomes

- Create and fit some basic multivariate time series models
- Learn how to fit some time series smoothing models: splines and Gaussian processes
- Understand the state of the art in time series models, including latent factor models and dynamic parameter models

Introduction to multivariate models

- Often we have multiple different observations at each time and we want to model them together
- For example, we might have multiple different climate variables observed, or multiple chemical signatures. If they are correlated then by fitting them separately we will lose precision
- If the observations are observed at different times for each time series we can use the NA trick, or create a latent state space model on which all measurements are regular

The Vector AR model

- We can extend most of the models we have met to multivariate scenarios by applying the multivariate normal distribution instead of the univariate version
- Suppose now that y_t is a vector of length k containing all the observations at time t
- We can write:

$$y_t = A + \Phi y_{t-1} + \epsilon_t,\ \epsilon_t \sim MVN(0, \Sigma)$$

or equivalently

$$y_t \sim MVN(A + \Phi y_{t-1}, \Sigma)$$

- where MVN is the multivariate normal distribution
- Here the parameter vector A controls the overall mean level for each of the k series, Φ is a $k \times k$ matrix which controls the influence on the current value of the previous time points of *both* series
- Σ here is a $k \times k$ matrix that controls the variance of the process and the residual correlation between the two series (NB: this is different from the GP formulation we met yesterday)

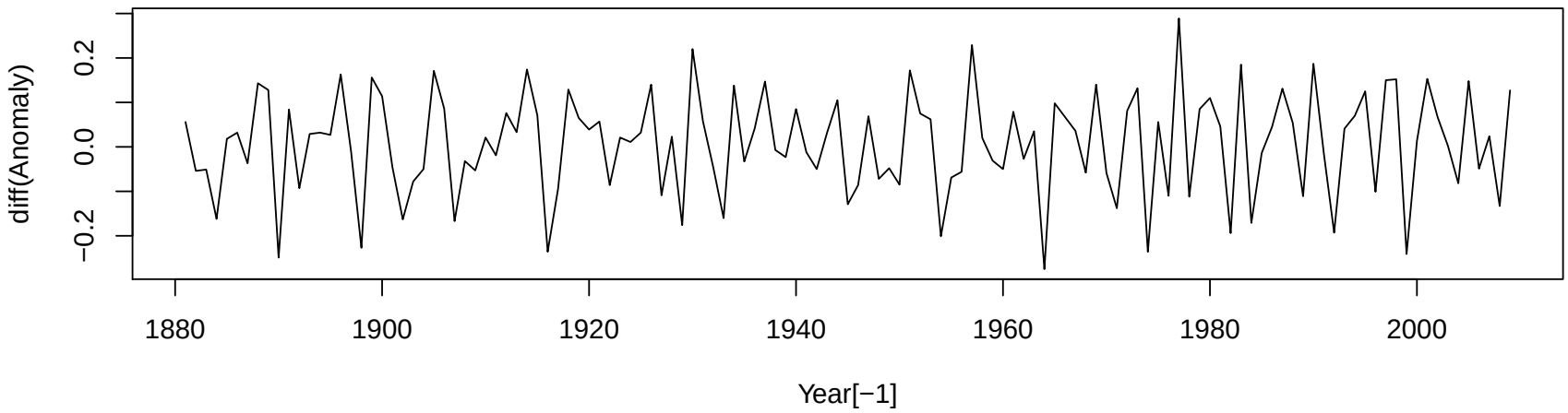
JAGS code for the VAR model

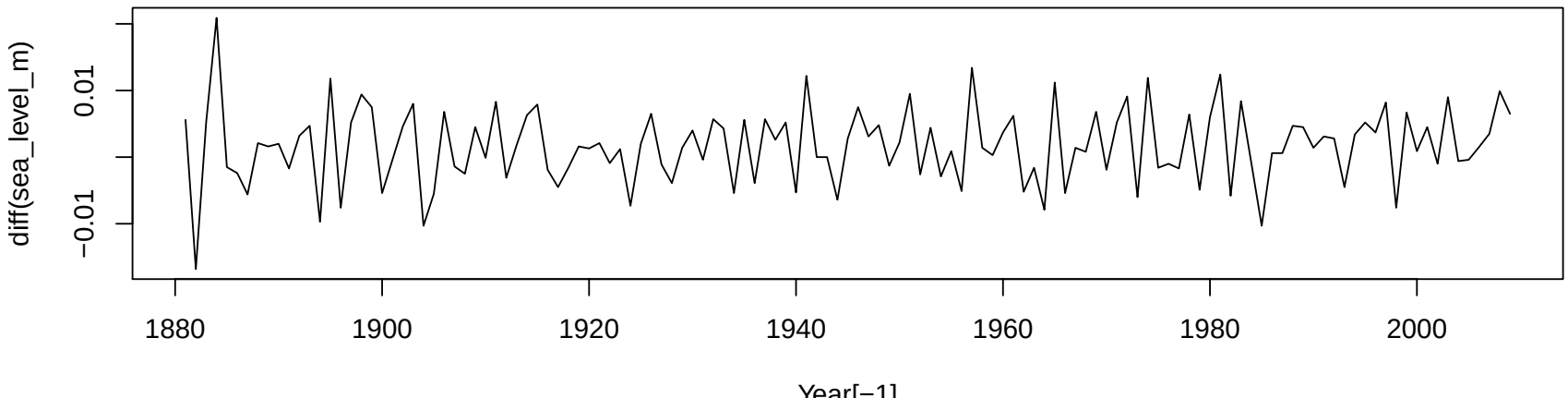
```
model_code = '
model
{
  # Likelihood
  for (t in 2:T) {
    y[t, ] ~ dmnorm(mu[t, ], Sigma.Inv)
    mu[t, 1:k] <- A + Phi %*% y[t-1,]
  }
  Sigma.Inv ~ dwish(I, k+1)
  Sigma <- inverse(Sigma.Inv)

  # Priors
  for(i in 1:k) {
    A[i] ~ dnorm(0, 0.01)
    Phi[i,i] ~ dunif(-1, 1)
    for(j in (i+1):k) {
      Phi[i,j] ~ dunif(-1,1)
      Phi[j,i] ~ dunif(-1,1)
```

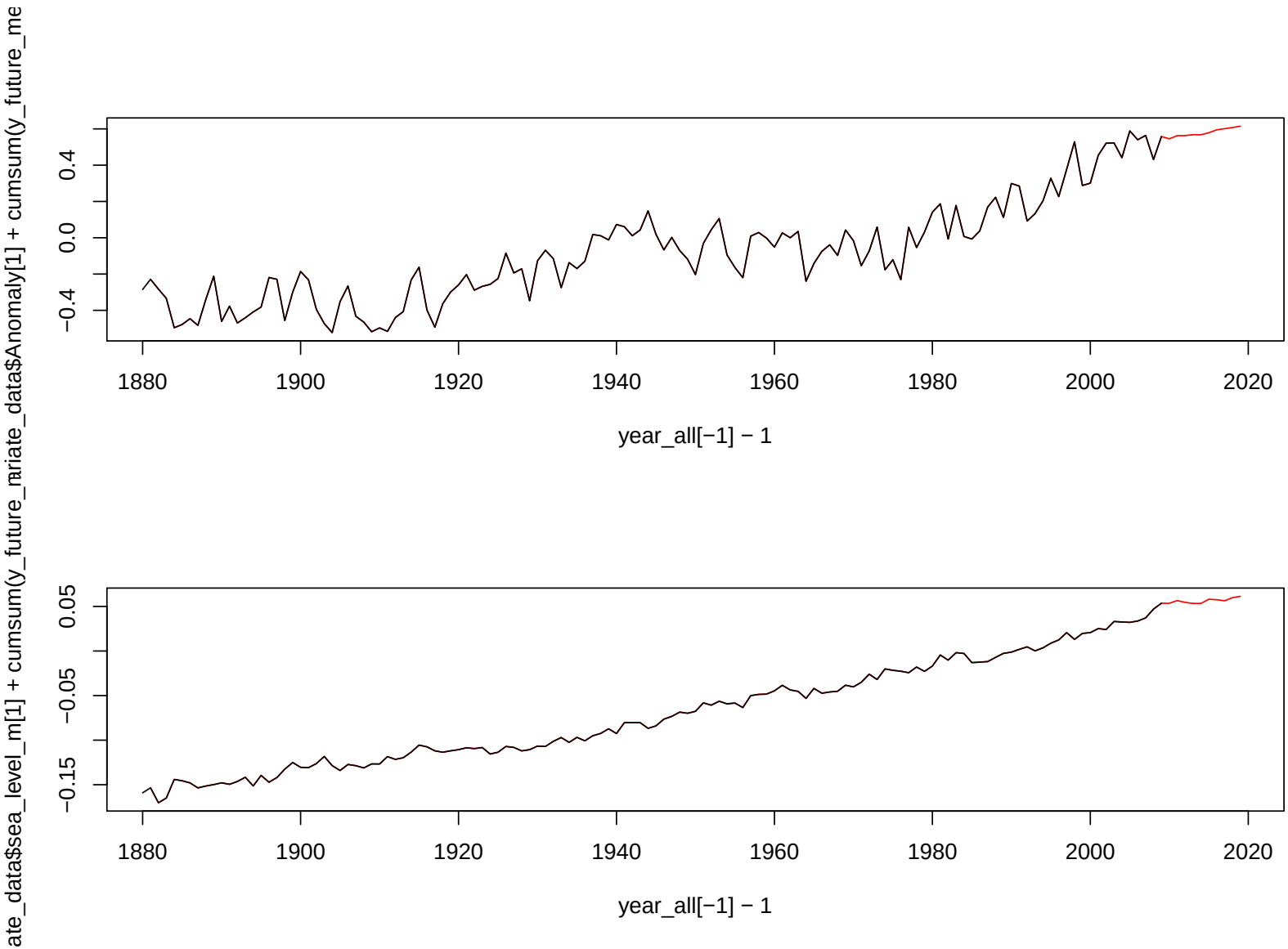
Example: joint temperature/sea level models

```
par(mfrow=c(2,1))
with(bivariate_data, plot(Year[-1], diff(Anomaly), type='l
with(bivariate_data, plot(Year[-1], diff(sea_level_m), type
```





VAR model results:



Splines and Gaussian processes

Regression with basis functions

- A common alternative to standard linear regression is to use polynomial regression:

$$y_i \sim N(\beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \ldots + \beta_p x_i^p, \sigma^2)$$

- This is simple to fit in JAGS/Stan using much of the code we have already seen
- However, when p is large this becomes very unwieldy, numerically unstable, hard to converge, and has some odd properties

Example

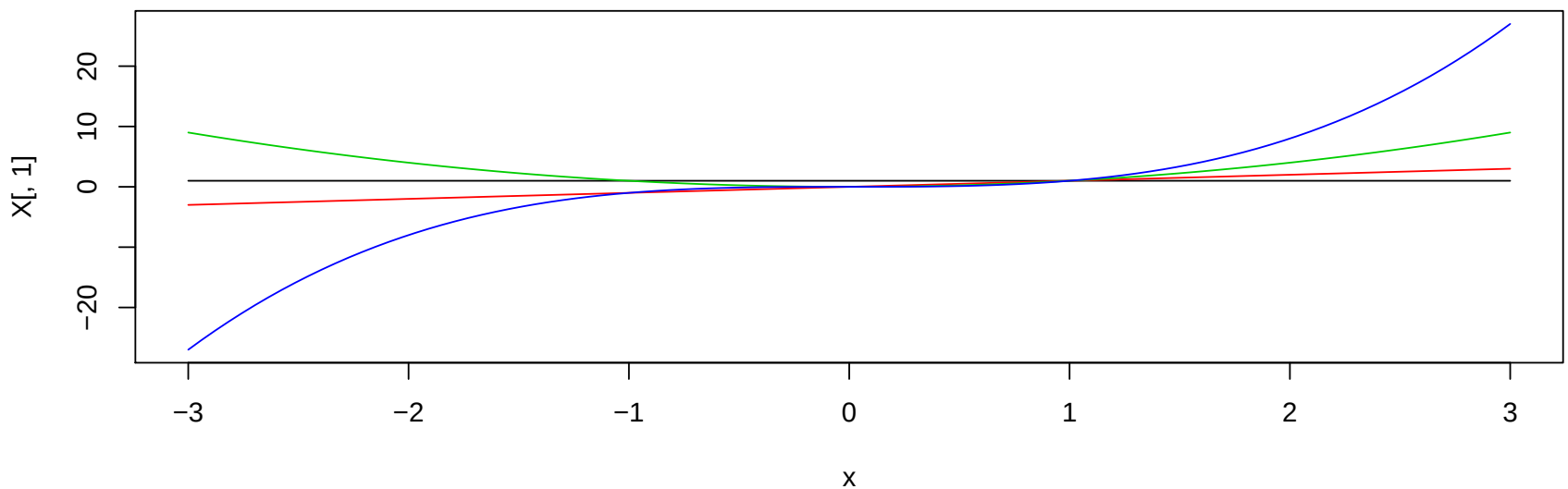
- Let's go back to the sheep data and fit a non-linear regression model:

```
sheep = read.csv('../../data/sheep.csv')
with(sheep, plot(year, sheep, type = 'l'))
```

Basis functions

- When you have a matrix X used in a regression model, the columns are often called *basis functions*

```
x = seq(-3, 3, length = 100)
X = cbind(1, x, x^2, x^3)
plot(x,X[,1],ylim=range(X), type = 'l')
for(i in 2:ncol(X)) lines(x,X[,i], col = i)
```



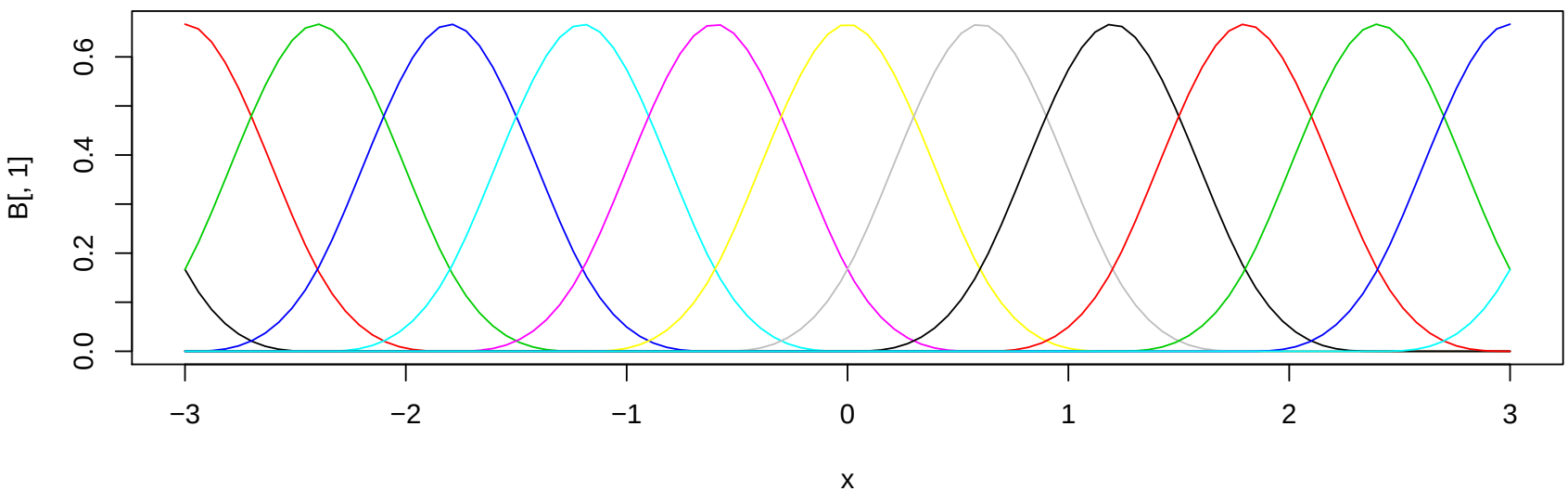
Creating new basis functions

- When we run a linear regression model using this matrix, we are multiplying each column by its associated β value, and forming an estimate of y
- As you can see, as we go up to powers of 3, 4, 5, etc, the values on the y-axis start to get really big
- Why not replace these *polynomial basis functions* with something better?

B-splines

- Here are some better, beautiful, basis functions called *B-spline* basis functions

```
B = bbase(x)
plot(x,B[,1],ylim=range(B), type = 'l')
for(i in 2:ncol(B)) lines(x,B[,i], col = i)
```



P-splines

- Now, instead of using a matrix X with polynomial basis functions, we create a matrix B of B-spline basis functions
- Each basis function gets its own weight β_j which determines the height of the curve
- A common way to make the curve smooth is make the β_j values similar to each other via a hierarchical model. Often:

$$\beta_j \sim N(\beta_{j-1}, \sigma_b^2)$$

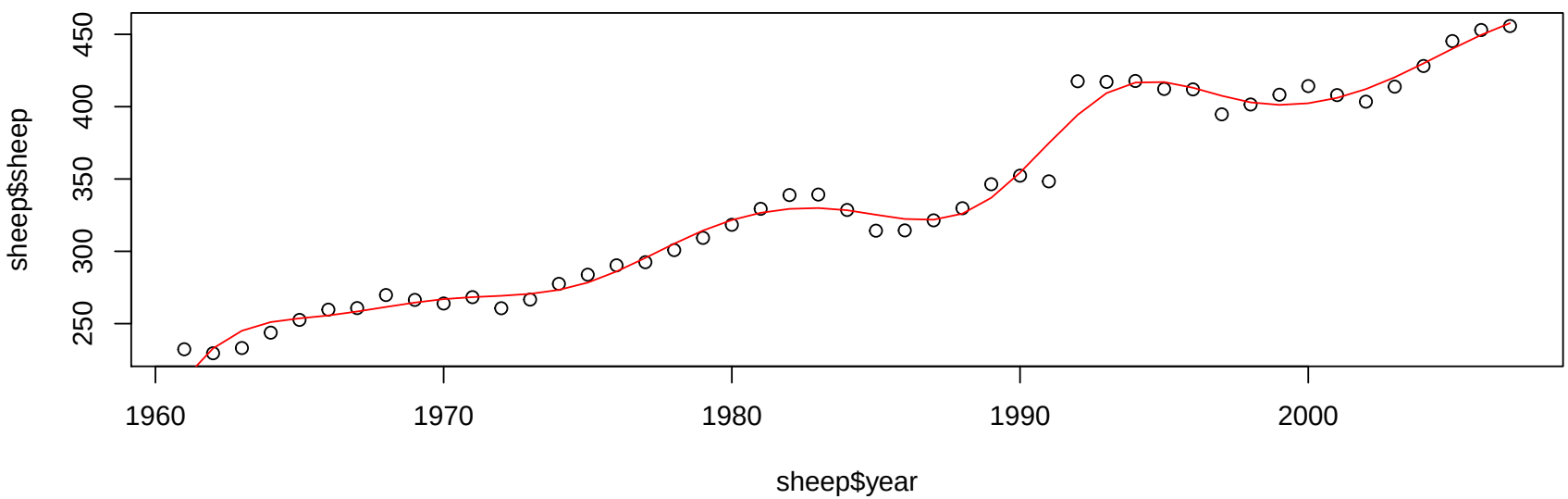
- This is known as a *penalised spline* or P-spline model since you are penalising the spline basis weights by making them similar to each other

Example code in Stan

```
stan_code = '
data {
  int<lower=0> N;
  int<lower=0> p;
  vector[N] y;
  matrix[N,p] B;
}
parameters {
  vector[p] beta;
  real<lower=0> sigma_b;
  real<lower=0> sigma;
}
model {
  y ~ normal(B * beta, sigma);
  beta[1] ~ normal(0, 10);
  for(j in 2:p)
    beta[j] ~ normal(beta[j-1], sigma_b);
  sigma_b ~ cauchy(0, 10);
  sigma ~ cauchy(0, 10);
}
'
```

Stan output

```
beta = apply(extract(stan_run, pars = 'beta')$beta,
             2,'mean')
plot(sheep$year, sheep$sheep)
lines(sheep$year, B%*%beta, col = 'red')
```



An alternative: Gaussian processes

- Whilst splines are great, they tend to use many parameters. You have to decide how many basis functions you want, and the advice is to use as many as possible
- The idea is that the penalty term induces the smoothness so it doesn't matter how many you used. But the model will be much slower to converge
- An alternative is to use a multivariate normal distribution, where you constrain the correlation between the elements of the distribution to be larger when they are close together. This is called a *Gaussian process*

A Gaussian process model:

- We write the Gaussian process model as:

$$y \sim MVN(\alpha, \Sigma + \sigma^2 I)$$

- Here α is a single parameter which represents the overall mean (but could also include regression covariates)
- Σ is a covariance matrix with terms:

$$\Sigma_{ij} = \tau^2 \exp\left(-\phi(x_i - x_j)^2\right)$$

  If you think about this, when x_i and x_j are close then you get a value of approximately τ^2. When they're far away you get a value of zero
- σ represents the residual standard deviation, as usual

A GP model in Stan

```
stan_code = '
data {
  int<lower=1> N;
  vector[N] x;
  vector[N] y;
}
transformed data {
  vector[N] alpha;
  for (i in 1:N) alpha[i] = 0;
}
parameters {
  real<lower=0> eta_sq;
  real<lower=0> inv_rho_sq;
  real<lower=0> sigma_sq;
}
transformed parameters {
  real<lower=0> rho_sq;
  rho_sq = inv(inv_rho_sq);
} model {
  matrix[N, N] Sigma;
  // off-diagonal elements
  for (i in 1:(N-1)) {
    for (j in (i+1):N) {
      Sigma[i, j] = eta_sq * exp(-rho_sq * pow(x[i] - x[j],2));
      Sigma[j, i] = Sigma[i, j];
    }
  }
  // diagonal elements
  for (k in 1:N)
    Sigma[k, k] = eta_sq + sigma_sq;
  eta_sq ~ cauchy(0, 5);
  inv_rho_sq ~ cauchy(0, 5);
  sigma_sq ~ cauchy(0, 5);
  y ~ multi_normal(alpha, Sigma);
}
'
```

Stan output

```
pred_ci = apply(extract(stan_run, pars = 'y2')$y2,
                2, 'quantile', c(0.25,0.5,0.75))
plot(sheep$year, sheep$sheep, xlim = c(min(sheep$year),
                                       max(sheep$year)+20),
     ylim = c(200, 550), type = 'l')
lines(x2, pred_ci[1,], col = 'red')
lines(x2, pred_ci[2,], col = 'red')
lines(x2, pred_ci[3,], col = 'red')
```

Notes on GPs

- Whilst GPs have far fewer parameters than splines, they tend to be slower to fit because the calculation of the density for the multivariate normal involves a matrix inversion which is really slow
- There are lots of fun ways to fiddle with GP models, as you can change the function that controls the way the covariance decays, or add in extra information in the mean
- There is a very useful but quite fiddly formula that enables you to predict for new values of y from new values of x just like a regression

Time Series: the state of the art

Mixing up state space models, multivariate time series, Gaussian processes

- We can extend the simple state space model we met earlier to work for multivariate series
- We would have a state equation that relates our observations to a multivariate latent time series (possibly of a different dimension)
- We could change the time series model of the latent state to be an ARIMA model, an O-U process, a Gaussian process, or anything else you can think of!

Dynamic linear models

- So far in all our models we have forced the time series parameters to be constant over time
- In a *Dynamic Linear Model* we have a state space model with :

$$y_t = F_t x_t + \epsilon_t,\ \epsilon_t \sim MVN(0, \Sigma_t)$$

$$x_t = G_t x_{t-1} + \gamma_t,\ \gamma_t \sim N(0, \Psi_t)$$

- The key difference here is that the transformation matrices F_t and G_t can change over time, as can the variance matrices Σ_t and Ψ_t, possibly in an ARCH/GARCH type framework
- These are very hard models to fit in jags but simple versions can work

Latent factor time series models

- If we have very many series, a common approach to reduce the dimension is to use Factor Analysis or Principal components
- In a latent factor model we write:

$$y_t = Bf_t + \epsilon_t$$

  where now B is a *numseries* $\times$ *numfactors* factor loading matrix which transforms the high dimensional y_t into a lower dimensional f_t.
- f_t can then be run using a set of univariate time series, e.g. random walks
- The B matrix is often hard to estimate and might require some tight priors

Summary

- We have seen how to fit basic Bayesian state space models and observed some of their pitfalls
- We know how to create some simple multivariate time series models
- We have seen some of the more advanced ideas in time series models such as DLMs and dynamic factor models
- You are now an expert in Bayesian time series!