Portfolio 3 - RcppArmadillo

Sam Bowyer

2023-05-29

Local Polynomial Regression

In this portfolio we'll use `RcppArmadillo` by looking at the following dataset on solar electricity production in Sidney, Australia[1].

```r
load("solarAU.RData")
head(solarAU)
```

```
##       prod          toy tod
## 8832 0.019 0.000000e+00   0
## 8833 0.032 5.708088e-05   1
## 8834 0.020 1.141618e-04   2
## 8835 0.038 1.712427e-04   3
## 8836 0.036 2.283235e-04   4
## 8837 0.012 2.854044e-04   5
```

Here, `prod` is the total production from 300 homes, `toy` $\in [0, 1]$ is the time-of-year and `tod` $\in \{0, 1, ..., 47\}$ is the time of day. We will be modelling the logarithm of `prod`, `logprod`, as a function of `toy` and `tod`, and to avoid problems when `prod` is 0, we'll add 0.01 to `logprod`.

```r
solarAU$logprod <- log(solarAU$prod+0.01)
```

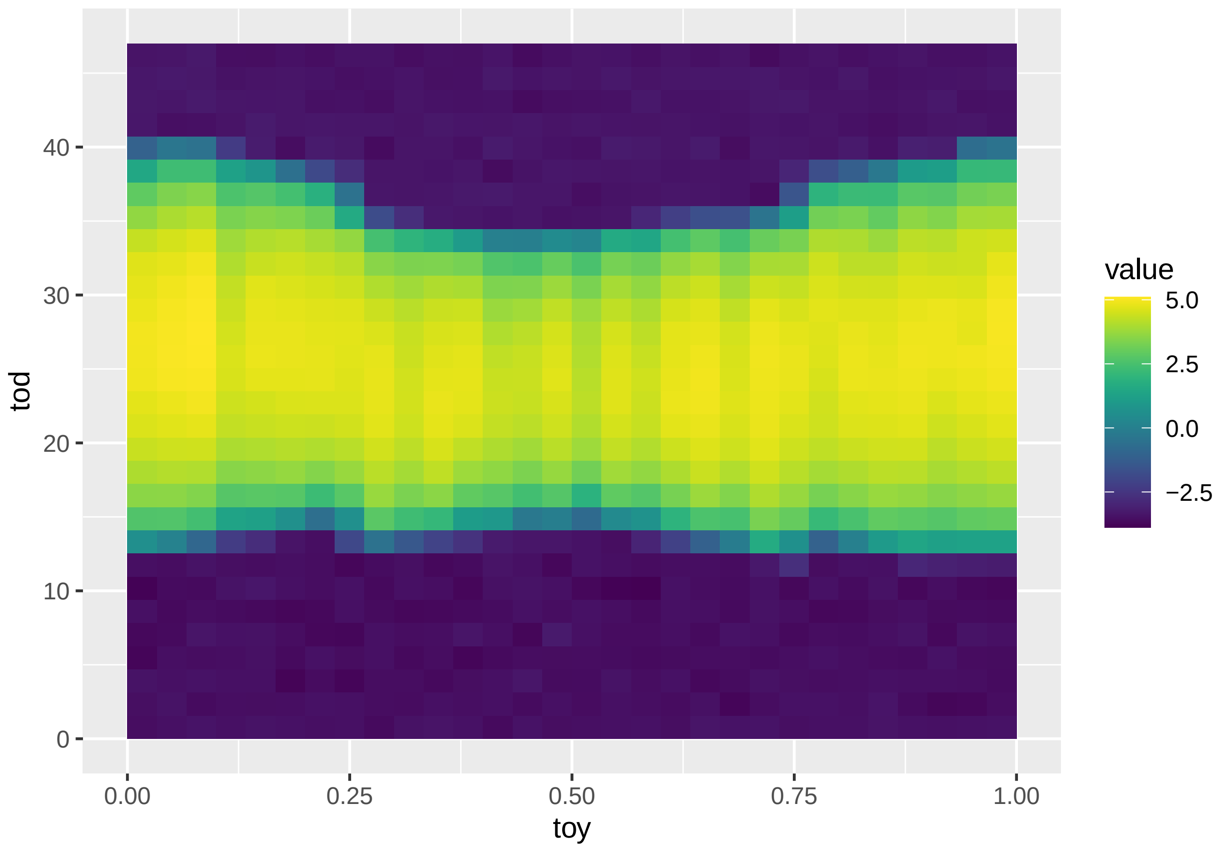
Having an initial look at the data, we see that production is, unsurprisingly, higher when there is more sunlight, i.e. in Australian winter (when `tod` is close to 0 or 1).

```r
library(ggplot2)
library(viridis)
```

```
## Loading required package: viridisLite
```

```r
ggplot(solarAU,
       aes(x = toy, y = tod, z = logprod)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))
```

[1]https://www.ausgrid.com.au/Industry/Our-Research/Data-to-share/Solar-home-electricity-data



First we consider a simple polynomial regression model

$$\mathbb{E}(y|\mathbf{x}) = \beta_0, +\beta_1 \text{tod} + \beta_2 \text{tod}^2 + \beta_3 \text{toy} + \beta_4 \text{toy}^2 = \tilde{\mathbf{x}}^T \beta$$

where $\tilde{\mathbf{x}} = \{\text{tod}, \text{tod}^2, \text{toy}, \text{toy}^2\}$. This can easily be done in R as follows:

```
fit <- lm(logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
```

However, we want to solve this faster by using `RcppArmadillo`.

Question 1

To find $\hat{\beta} = \text{argmin}_\beta \, ||\mathbf{y} - \mathbb{X}\beta||^2$ we need to compute $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$, but the matrix inversion therein can be numerically unstable, we we'll write $\mathbf{X}$ using the QR decomposition, that is:

$$\mathbf{X} = \mathbf{QR} = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} = \mathbf{Q}_1\mathbf{R}_1 \in \mathbb{R}^{m \times n}$$

where $\mathbf{R}_1 \in \mathbb{R}^{n \times n}$ is upper triangular, $\mathbf{0} \in \mathbb{R}^{(m-n) \times n}$ is a zero matrix, and both $\mathbf{Q}_1 \in \mathbb{R}^{m \times n}$ and $\mathbf{Q}_2 \in \mathbb{R}^{m \times (m-n)}$ have orthogonal columns. In this case our coefficients $\hat{\beta}$ become:

$$\hat{\beta} = \mathbf{R}_1^{-1}\mathbf{Q}_1^T\mathbf{y}$$

.

Below we create our design matrix X.

```
X = with(solarAU, cbind(1, tod, tod^2, toy, toy^2))
y = solarAU$logprod
head(X); head(y)
```

```
##        tod             toy
## [1,] 1   0  0 0.000000e+00 0.000000e+00
## [2,] 1   1  1 5.708088e-05 3.258227e-09
## [3,] 1   2  4 1.141618e-04 1.303291e-08
## [4,] 1   3  9 1.712427e-04 2.932405e-08
## [5,] 1   4 16 2.283235e-04 5.213164e-08
## [6,] 1   5 25 2.854044e-04 8.145568e-08
```

```
## [1] -3.540459 -3.170086 -3.506558 -3.036554 -3.079114 -3.816713
```

Next we write our `Rcpp` version of `lm` using the QR decomposition function `qr_econ`. In this we also speed up copmutation by specifying that R is upper-triangular (with `trimatu(R)`) and setting `solve_opts::fast`.

```
library(Rcpp)
library(RcppArmadillo)

sourceCpp(code = '
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace arma;

// [[Rcpp::export(name = "lm_Rcpp")]]
vec lm_Rcpp(mat& X, vec& y){
  mat Q, R;

  qr_econ(Q, R, X);

  return solve(trimatu(R), Q.t() * y, solve_opts::fast);
}
')

fit_Rcpp = lm_Rcpp(X, y)
t(fit_Rcpp); fit$coefficients
```

```
##           [,1]      [,2]        [,3]      [,4]     [,5]
## [1,] -6.262757 0.8644039 -0.01757599 -5.918069 6.142989
```

```
## (Intercept)         tod    I(tod^2)         toy    I(toy^2)
## -6.26275685  0.86440391 -0.01757599 -5.91806924  6.14298863
```

Clearly the `Rcpp` implementation is reaching the same results, but we can check and see that it is in fact faster than using `lm`:

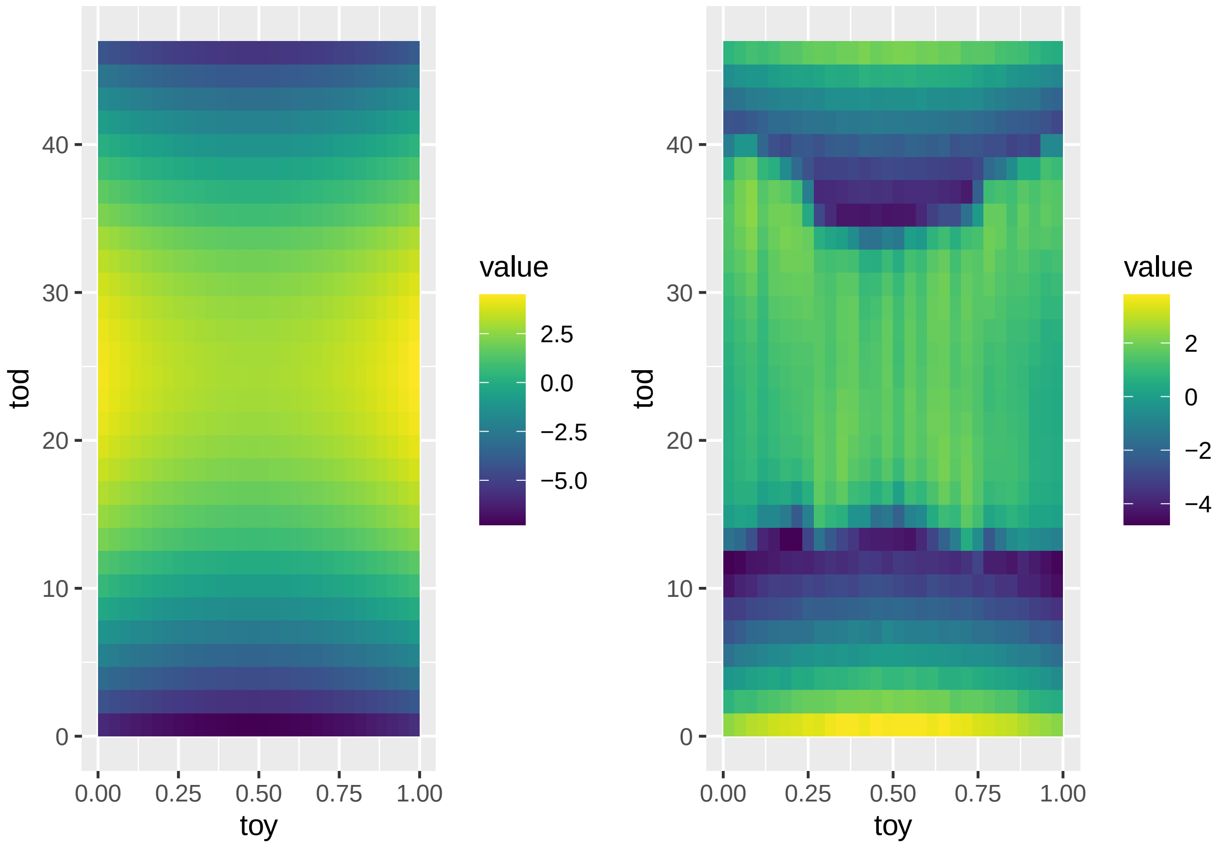
```
library(microbenchmark)

lm_ <- function() lm(logprod ~ tod + I(tod^2) + toy + I(toy^2), data = solarAU)
lm_Rcpp_ <- function() lm_Rcpp(X, y)
microbenchmark(lm_(), lm_Rcpp_(), times = 1000)
```

```
## Unit: microseconds
##        expr      min        lq      mean    median       uq       max neval
##       lm_() 1635.450 1772.4745 2516.3336 1888.1300 2232.813 53066.581  1000
##  lm_Rcpp_()  409.649  420.8865  476.1629  467.0285  503.654  1253.266  1000
```

Question 2

Unfortunately, looking at the polynomial fit we can see that there is a non-linear pattern left in the residuals:

```
library(gridExtra)

solarAU$fitPoly <- fit$fitted.values

pl1 <- ggplot(solarAU,
              aes(x = toy, y = tod, z = fitPoly)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))

pl2 <- ggplot(solarAU,
              aes(x = toy, y = tod, z = logprod - fitPoly)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))
grid.arrange(pl1, pl2, ncol = 2)
```



To improve our fit then, we'll use a local least regression model in which the estimated regression coefficients depend on $\mathbf{x}$, i.e. $\hat{\beta} = \hat{\beta}(\mathbf{x})$. In this case, we find $\hat{\beta}(\mathbf{x}_0)$ as

$$\hat{\beta}(\mathbf{x}_0) = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} \kappa_{\mathbf{H}}(\mathbf{x}_0 - \mathbf{x}_i)(y_i - \tilde{\mathbf{x}}_i^T \beta)^2$$

where $\kappa_{\mathbf{H}}$ is a density kernel with positive definite bandwidth matrix $\mathbf{H}$. Since $\kappa_{\mathbf{H}}(\mathbf{x}_0 - \mathbf{x}_i) \to 0$ as $||\mathbf{x}_0 - \mathbf{x}_i|| \to \infty$, we know that $\hat{\beta}(\mathbf{x}_0)$ will depend more on data points closer to $\mathbf{x}_0$ than those far away.

In `R` we can do this using the Gaussian kernel as follows:

```
library(mvtnorm)
lmLocal <- function(y, x0, X0, x, X, H){
  w <- dmvnorm(x, x0, H)
  fit <- lm(y ~ -1 + X, weights = w)
  return( t(X0) %*% coef(fit) )
}
```

But rather than re-estimating the model for every $\mathbf{x}_0$ (of which there are 17472), we will only use 2000 of the rows in our data, sampled uniformly at random without replacement.

```
nrow(solarAU)
```

```
## [1] 17472
```

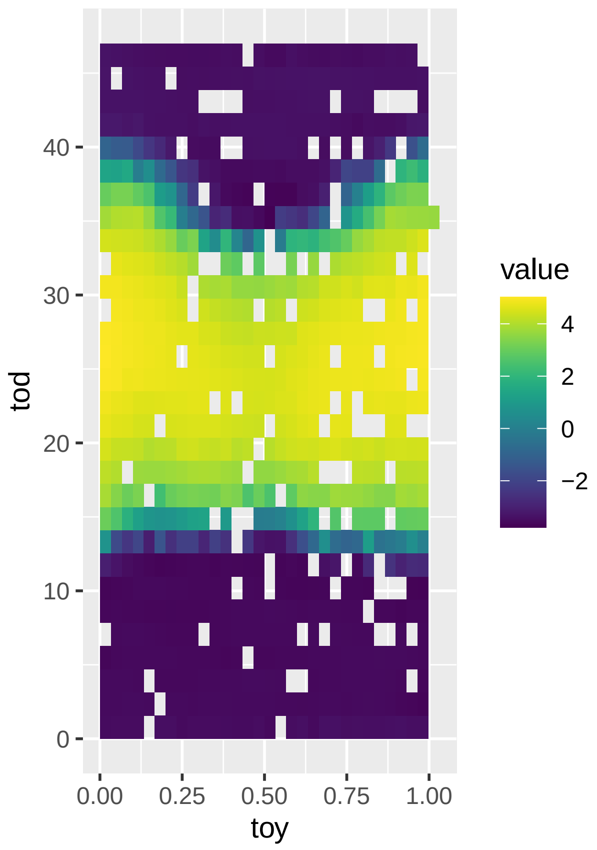
```
n <- nrow(X)
nsub <- 2e3
sub <- sample(1:n, nsub, replace = FALSE)

y <- solarAU$logprod
solarAU_sub <- solarAU[sub, ]
x <- as.matrix(solarAU[c("tod", "toy")])
x0 <- x[sub, ]
X0 <- X[sub, ]
```

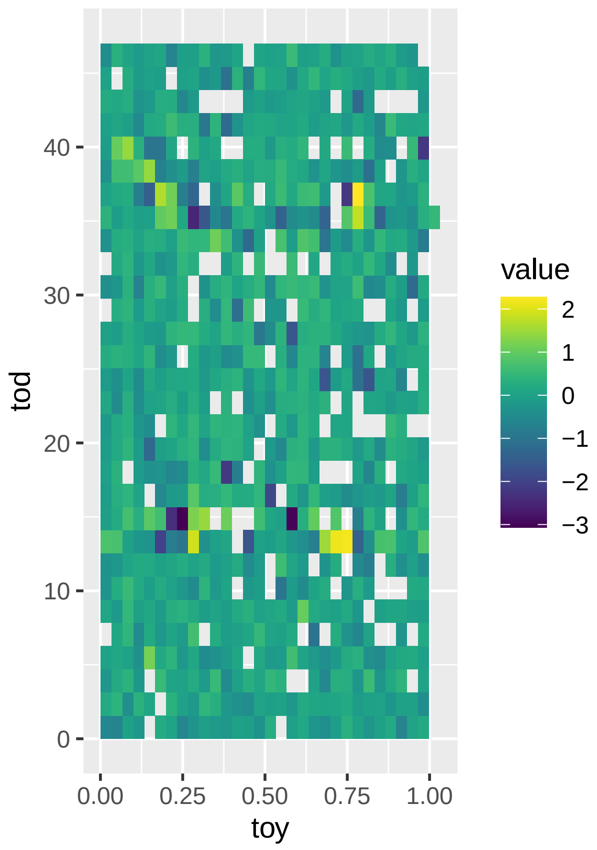
And we obtain estimates at each of these locations as follows:

```
predLocal <- sapply(1:nsub, function(ii){
  lmLocal(y = y, x0 = x0[ii, ], X0 = X0[ii, ], x = x, X = X, H = diag(c(1, 0.1)^2))
})
```

This leads to a better fit with no clear pattern in the residuals left over.

```
solarAU_sub$fitLocal <- predLocal

pl1 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = fitLocal)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))

pl2 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = logprod - fitLocal)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))

grid.arrange(pl1, pl2, ncol = 2)
```





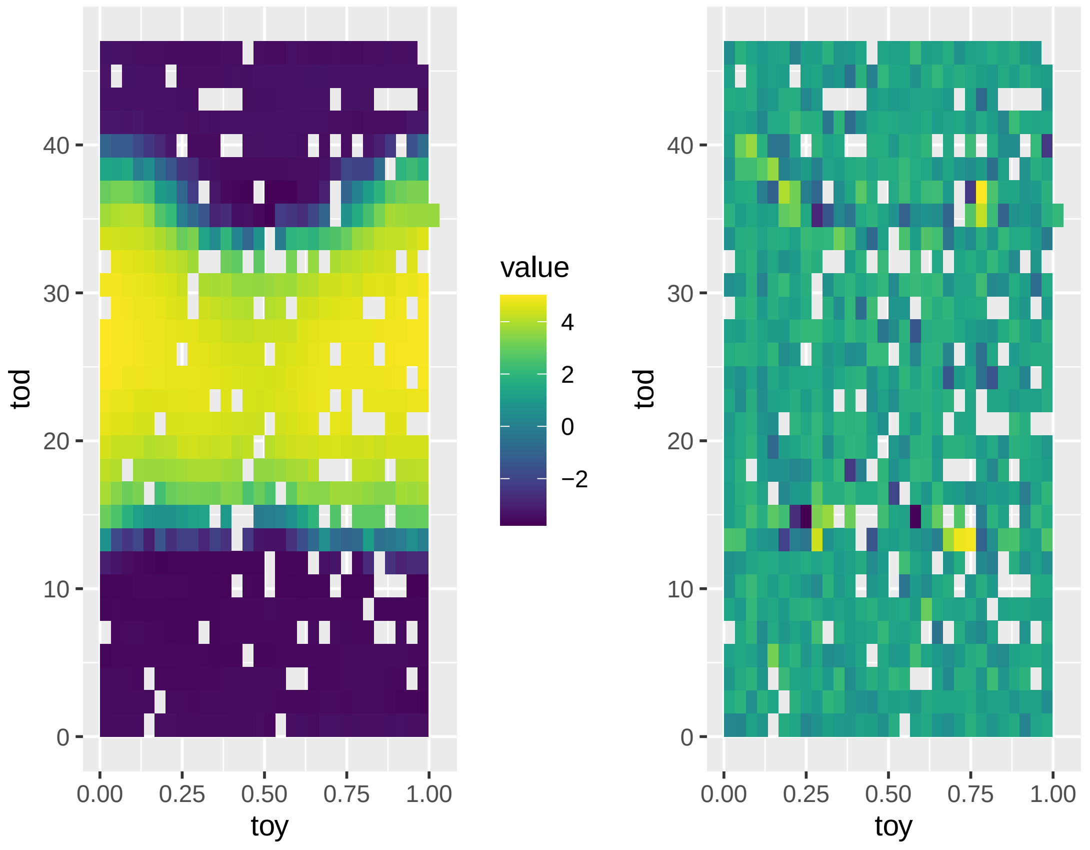
However, this is extremely slow, so we will speed this up by implementing it in `RcppArmadillo` (we have to implement the Gaussian kernel in here too).

```
sourceCpp(code = '
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace arma;

vec dmvnorm_(mat & X, const rowvec & mu, mat & L){
  // Gaussian density kernel

  unsigned int m = X.n_rows;
  unsigned int n = X.n_cols;

  vec D = L.diag();

  vec out(m);
  vec z(n);

  double acc;

  unsigned int icol, irow, ii;

  for(icol = 0; icol < m; icol++){
    for(irow = 0; irow < n; irow++){
      acc = 0.0;
      for(ii = 0; ii < irow; ii++){
          acc += z.at(ii) * L.at(irow, ii);
        }
        z.at(irow) = ( X.at(icol, irow) - mu.at(irow) - acc ) / D.at(irow);
      }
      out.at(icol) = sum(square(z));
    }
    out = exp( - 0.5 * out - ( (n / 2.0) * log(2.0 * M_PI) + sum(log(D)) ) );

    return out;
}


vec lm(mat X, vec y){
  // fit lm with QR decomp as before

  mat Q, R;

  qr_econ(Q, R, X);

  return solve(trimatu(R), Q.t() * y, solve_opts::fast);
}

// [[Rcpp::export(name = "lm_local_Rcpp")]]
vec lm_local(vec& y, mat& x0, mat& X0, mat& x, mat& X, mat& H){
  // find coefficients for each row and weight by the Gaussian density kernel

  int nsub = x0.n_rows;
  vec out(nsub), weights;
  double fit;

  mat L = chol(H, "lower");

  for(int i=0; i<nsub; i++){
    weights = sqrt(dmvnorm_(x, x0.row(i), L));

    fit = as_scalar(X0.row(i) * lm(X.each_col() % weights, y % weights));

    out(i) = fit;
  }

  return out;
}
')
```

Fitting this RcppArmadillo version we obtain the same results as before.

```r
predLocal_Rcpp <- lm_local_Rcpp(y, x0, X0, x, X, diag(c(1, 0.1)^2))

solarAU_sub$fitLocal <- predLocal_Rcpp

pl1 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = fitLocal)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))
```

```r
pl2 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = logprod - fitLocal)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))

grid.arrange(pl1, pl2, ncol = 2)
```



```r
all.equal(as.vector(predLocal_Rcpp), predLocal)
```

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

But, as expected, the `RcppArmadillo` version is much faster.

```r
library(microbenchmark)

local_R <- function() {
predLocal <- sapply(1:nsub, function(ii){
  lmLocal(y = y, x0 = x0[ii, ], X0 = X0[ii, ], x = x, X = X, H = diag(c(1, 0.1)^2))
  })
}
local_C <- function() lm_local_Rcpp(y, x0, X0, x, X, diag(c(1, 0.1)^2))
microbenchmark(local_R(), local_C(), times=10)
```

```
## Unit: seconds
##       expr       min       lq      mean    median        uq       max neval
##  local_R() 11.196285 11.37293 11.495952 11.538341 11.656982 11.713108    10
##  local_C()  1.738081  1.78699  2.754891  2.132787  4.121924  4.146339    10
```

Question 3

Finally, we can implement a cross-validation routine to choose the bandwidth matrix $\mathbf{H}$ (which we will keep as diagonal) by minimising the residual sum of squares. In particular, we'll test all combinations of the diagonal elements of $\mathbf{H}$ being in $\{0.01, 0.1, 1, 10, 100\}$.

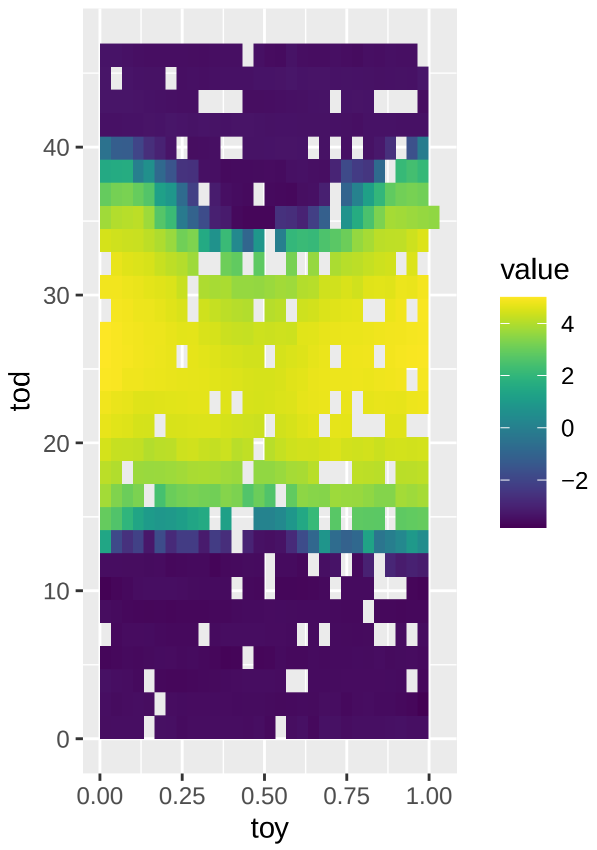
```
H_elems = c(0.01, 0.1, 1, 10, 100)
opt_RSS = Inf
opt_H_elems = c(NA, NA)

for (H_11 in H_elems){
  for (H_22 in H_elems){
    fit = lm_local_Rcpp(y, x0, X0, x, X, diag(c(H_11, H_22)))
    RSS = sum((fit - solarAU_sub$logprod)^2)
    if (RSS < opt_RSS){
      opt_RSS = RSS
      opt_H_elems = c(H_11, H_22)
    }
  }
}
```

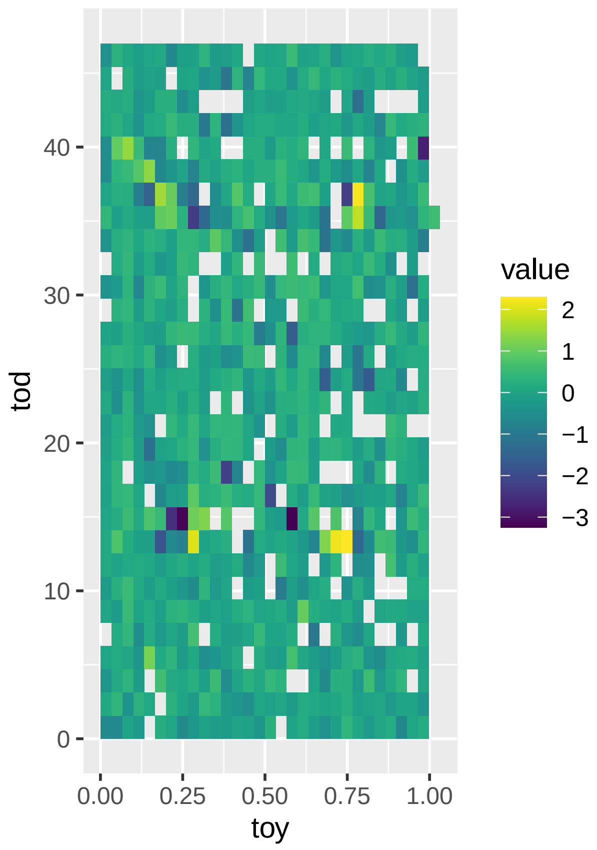
```
opt_H_elems
```

```
## [1] 0.01 0.01
```

Here we find that the optimal bandwidth matrix we tried was $\mathbf{H} = \begin{pmatrix} 0.01 & 0 \\ 0 & 0.01 \end{pmatrix}$, which gives us the following fit and residuals:

```
predLocal_Rcpp <- lm_local_Rcpp(y, x0, X0, x, X, diag(opt_H_elems))

solarAU_sub$fitLocal <- predLocal_Rcpp

pl1 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = fitLocal)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))

pl2 <- ggplot(solarAU_sub,
       aes(x = toy, y = tod, z = logprod - fitLocal)) +
       stat_summary_2d() +
       scale_fill_gradientn(colours = viridis(50))

grid.arrange(pl1, pl2, ncol = 2)
```


tod
0
10
20
30
40
toy
0.00
0.25
0.50
0.75
1.00
value
4
2
0
−2


tod
0
10
20
30
40
toy
0.00
0.25
0.50
0.75
1.00
value
2
1
0
−1
−2
−3