UNIVERSITY OF
COPENHAGEN

Parallelization and Scatterplot Smoothing

Computational Statistics

Johan Larsson

Department of Mathematical Sciences, University of Copenhagen

March 31, 2026

Today

Parallelization
Using multiple cores to speed up computations.

Scatterplot Smoothing
Cover **nearest neighbor smoothing** and **smoothing splines**

Parallelization

Parallelization

Most processors have multiple cores.

But unless instructed otherwise, only a single core is going to be used.

The computer doesn't automatically know that your computations are safe to do in parallel.

Embarrassingly Parallel Tasks

Tasks that easily separate into independent subtasks.

Examples

- Summing a vector (or matrix): `sum()`
- Linear algebra operations: `%*%` (`crossprod()`)
- Running iterations of a simulation
- Cross-validation

The foreach Package

```
library(foreach)
```

```
foreach(i = seq_len(3)) %dopar%
  {
    sqrt(i)
  }
```

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

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
```

Notes

- Returns a list (so not really a for loop).
- Nothing is actually parallel yet!
- First, we need to register a **backend**.

Backends

Multiple backends, installed separately (**doParallel**, **doMC**, **doFuture**) Load and **register** one before using foreach.

```
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)

foreach(i = seq_len(3)) %dopar%
  {
    paste(
      "Hello from process",
      Sys.getpid()
    )
  }
```

```
[[1]]
[1] "Hello from process 10397"

[[2]]
[1] "Hello from process 10398"

[[3]]
[1] "Hello from process 10397"
```

Combining Results

`foreach()` always returns a list.

If you want to reduce your result, use either the `.combine` argument or manually reduce the resulting list.

```
x <- c(4, 1, 1e2, 2)

foreach(i = seq_len(4), .combine = c) %dopar%
  {
    log(x[[i]])
  }
```

```
[1] 1.3862944 0.0000000 4.6051702 0.6931472
```

Futures

A future represents a value that may be available later on.

Sometimes called promises: "I promise to give you the result later."

Without Futures

```
v <- {
  cat("Hello world!\n")
  3.14
}
```

```
Hello world!
```

```
v
```

```
[1] 3.14
```

With Futures

```
library(future)
v %<-%
  {
    cat("Hello world!\n")
    3.14
  }
v
```

```
Hello world!
```

```
[1] 3.14
```

Futures Enable Parallelization

A new thread is created for each future.

- Meanwhile, the main thread can continue doing other work.
- Once the value is needed, the main thread blocks until it's available.

```
plan(multisession) # or multicore, mirai_multisession, etc.

z %<-%
  {
    # Peform some expensive computation
  }

# Perform other work here ...

z # This line blocks until z is available
```

Backends for Futures

Need to pick a backend (plan), with multiple options.

`multicore`
Available on Linux and macOS, but not Windows or RStudio. *Forks* the current R process, instead of starting a new one. But *not* recommended because of potential stability issues.

`mirai_multisession/mirai_cluster`
A newer alternative, available through the future.mirai package.

Caveats

Realistic Expectations
In practice, twice as many cores $\neq$ twice as fast (in practice).

Overhead
Parallelization comes with overhead (starting threads, communication).

Thread Safety
Some functions are not thread safe (e.g. random number generation).

Redundancies
Parallelizing a task that is already parallelized may not help and may even hurt performance.

Comparison of foreach and future

foreach
Main use case is to run computations in parallel that would otherwise be done in a `for` loop.

future
More general abstraction for asynchronous programming and parallelization.

Primarily used when you have various independent tasks that you want to run in parallel.

Scatterplot Smoothing

Nuuk Temperature Data

```
nuuk_plot <- ggplot(nuuk_year, aes(Year, Temperature)) +
  geom_point()
nuuk_plot
```

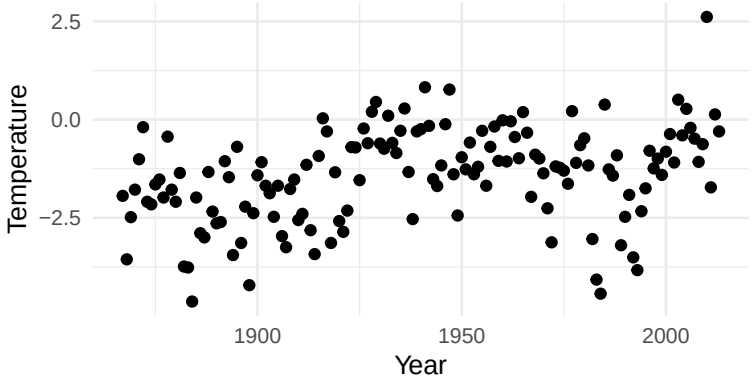


Figure 1: Annual average temperature in Nuuk, Greenland

Data is available here

Nearest Neighbor Estimation

Given data points $(x_1, y_1), \ldots, (x_n, y_n)$, the k-nearest neighbor estimator at x_i is

$$\hat{f}_i = \frac{1}{k} \sum_{j \in N_i} y_j$$

where N_i is the set of indices for the k nearest neighbors of x_i.

This is an estimator of

$$f(x) = E(Y \mid X = x).$$

Matrix Formulation

We can express all fitted values as a matrix-vector product:

$$\hat{\boldsymbol{f}} = \boldsymbol{S}\boldsymbol{y}$$

where $\boldsymbol{y} = (y_1, \dots, y_n)^T$ and $\boldsymbol{S}$ is the **smoother matrix**.

Smoother Matrix
Each entry of $\boldsymbol{S}$ is

$$S_{ij} = \frac{1}{k} \cdot \mathbf{1}_{\{j \in N_i\}}$$

where $\mathbf{1}_{\{j \in N_i\}}$ is 1 if j is a neighbor of i, 0 otherwise.

Linear Smoothers

Smoothers of the form $\hat{\boldsymbol{f}} = \boldsymbol{S}\boldsymbol{y}$ are called a *linear* smoothers.

The k nearest neighbor smoother is a simple example of a linear smoother that works for x-values in any metric space.

The representation of a linear smoother as a matrix-vector product,

$$\boldsymbol{S}\boldsymbol{y}$$

is theoretically useful, but often not the best way to actually compute $\hat{\boldsymbol{f}}$.

Running Mean

When $x_i \in \mathbb{R}$ we can sort data according to x-values and then use a *symmetric* neighbor definition:

$$N_i = \left\{ i - \frac{k-1}{2}, i - \frac{k-1}{2} + 1, \ldots, i-1, i, i+1, \ldots, i + \frac{k-1}{2} \right\}$$

(for k odd.)

A version of nearest neighbor smoothing that assumes ordered and symmetric neighborhoods.

Simplifies Computations

We don't need to keep track of metric comparisons. Only the order matters.

Running Mean (Naive Implementation)

Assume y is sorted and k is odd.

```
run_mean_naive <- function(y, k) {
  n <- length(y)
  m <- (k - 1) / 2
  y <- y / k
  f_hat <- rep(NA, n)

  for (i in (m + 1):(n - m - 1)) {
    f_hat[i] <- mean(y[(i - m):(i + m)])
  }

  f_hat
}
```

Efficient Running Mean

We can achieve a more efficient implementation by noting that

$$\hat{f}_{i+1} = \hat{f}_i - \frac{y_{i-(k-1)/2}}{k} + \frac{y_{i+(k+1)/2}}{k}.$$

```
run_mean <- function(y, k) {
  n <- length(y)
  m <- floor((k - 1) / 2)
  k <- 2 * m + 1 # Ensures k is odd and m = (k - 1) / 2
  y <- y / k
  f_hat <- rep(NA, n)
  f_hat[m + 1] <- sum(y[1:k])

  for (i in (m + 1):(n - m - 1)) {
    f_hat[i + 1] <- f_hat[i] - y[i - m] + y[i + 1 + m]
  }
  f_hat
}
```

```
f_hat <- run_mean(nuuk_year$Temperature, 11)
nuuk_plot + geom_line(aes(y = f_hat), color = "blue")
```

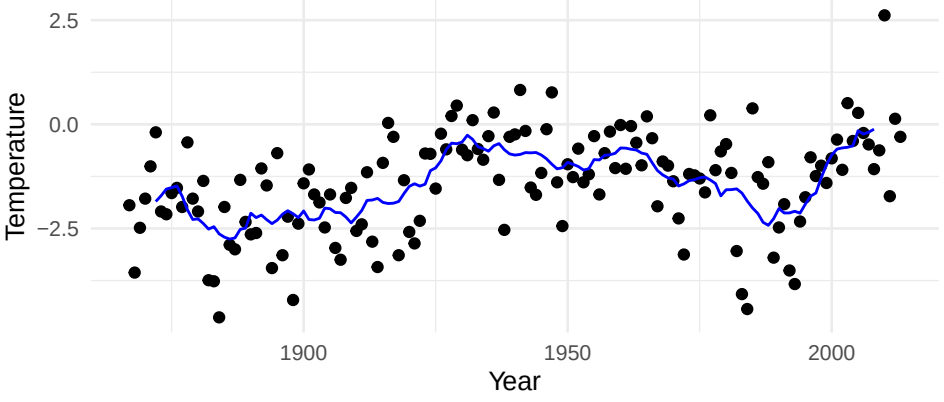


Figure 2: Running mean smoother with $k = 11$.

stats::filter()

`stats::filter()` applies linear filtering (moving averages or autoregression).

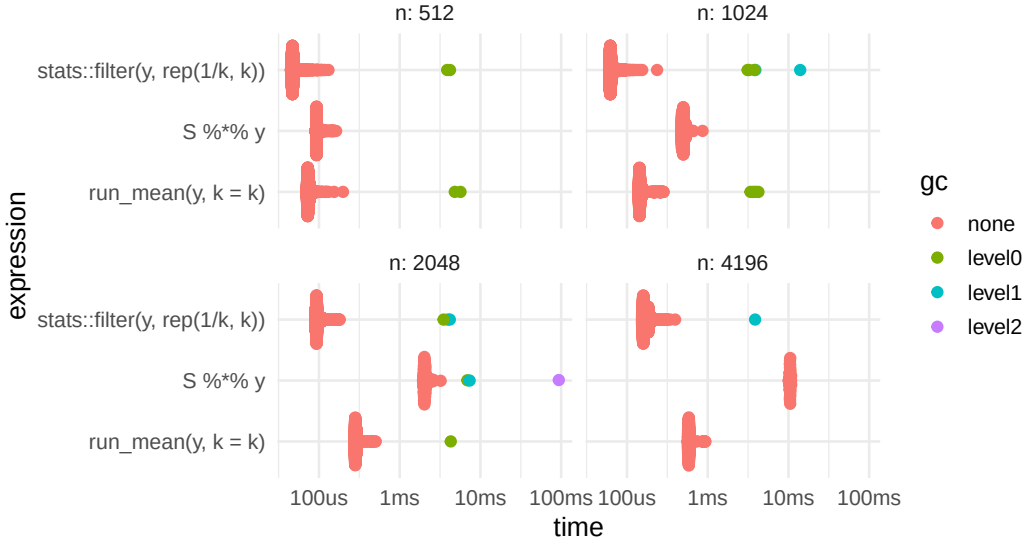
```
f_hat_filter <- stats::filter(
  nuuk_year$Temperature,
  rep(1 / 11, 11)
)
```

```
 [1]    NA    NA    NA    NA    NA -1.85 -1.72 -1.55 -1.52
[10] -1.48 -0.55 -0.52 -0.15 -0.23 -0.19 -0.12    NA    NA
[19]    NA    NA    NA
```

```
f_hat <- run_mean(nuuk_year$Temperature, 11)

all.equal(f_hat, as.numeric(f_hat_filter))
```

```
[1] TRUE
```



The Matrix-vector multiplication is $O(n^2)$.

The two other algorithms are $O(n)$.

There is a bias-variance trade-off in nearest neighbor smoothing with respect to k.

If data is i.i.d. with $V(Y \mid X) = \sigma^2$ and $f(x) = E(Y \mid X = x)$,

$$\begin{aligned}
\text{MSE}(x) &= \mathrm{E}(f(x) - \hat{f}(x))^2 \\
&= \mathrm{Var}\left(f(x) - \hat{f}(x)\right) + \left(\mathrm{E}(f(x) - \hat{f}(x))\right)^2 \\
&= \underbrace{\frac{\sigma^2}{k}}_{\text{variance}} + \underbrace{\left(f(x) - \frac{1}{k}\sum_{l \in N(x)} f(x_l)\right)^2}_{\text{bias}}.
\end{aligned}$$

Bias-Variance Trade-Off

- Small k gives large variance and small bias (if f is smooth).
- Large k gives small variance and potentially large bias (if f is not constant).

How to Choose k?

We know there is a bias-variance trade-off in k—so how do we choose k?

Leave-One-Out Cross Validation (LOOCV)

Leave out one observation at a time, predict using remaining data, and measure the prediction error.

The general LOOCV formula is

$$\text{LOOCV} = \sum_{i=1}^{n} (y_i - \hat{f}_i^{-i})^2.$$

This leads to the simplified formula

$$\text{LOOCV} = \sum_{i=1}^{n} \left(\frac{y_i - \hat{f}_i}{1 - S_{ii}} \right)^2.$$

For a *linear* smoother, we have

$$\hat{f}_i^{-i} = \sum_{j \neq i} \frac{S_{ij} y_j}{1 - S_{ii}},$$

LOOCV for Nearest Neighbors

For nearest neighbors, $S_{ii} = \frac{1}{k}$ for all i, so

$$\text{LOOCV} = \sum_{i=1}^{n} \left(\frac{y_i - \hat{f}_i}{1 - 1/k} \right)^2.$$

Implementation

```
loocv <- function(k, y) {
  f_hat <- run_mean(y, k)
  mean(((y - f_hat) / (1 - 1 / k))^2, na.rm = TRUE)
}
```

Note

The implementation removes missing values due to the way we handle the boundaries, and it uses `mean()` instead of `sum()` to correctly adjust for this.

```r
k <- seq(3, 40, 2)
cv_error <- sapply(k, function(kk) loocv(kk, nuuk_year$Temperature))
k_opt <- k[which.min(cv_error)]
```

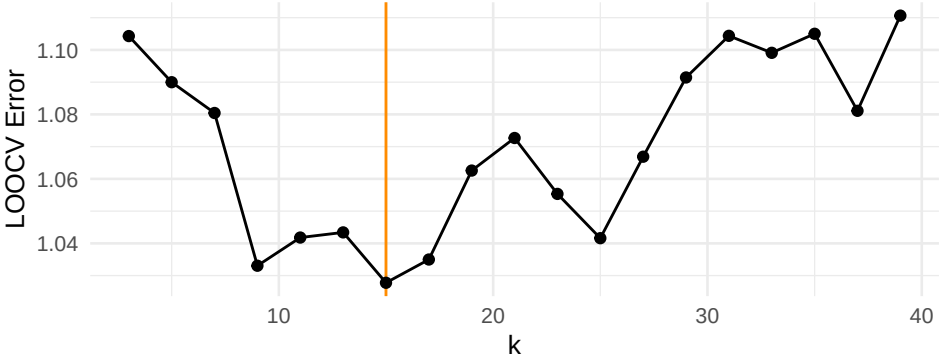


Figure 3: LOOCV error for nearest neighbor smoothing as a function of k.

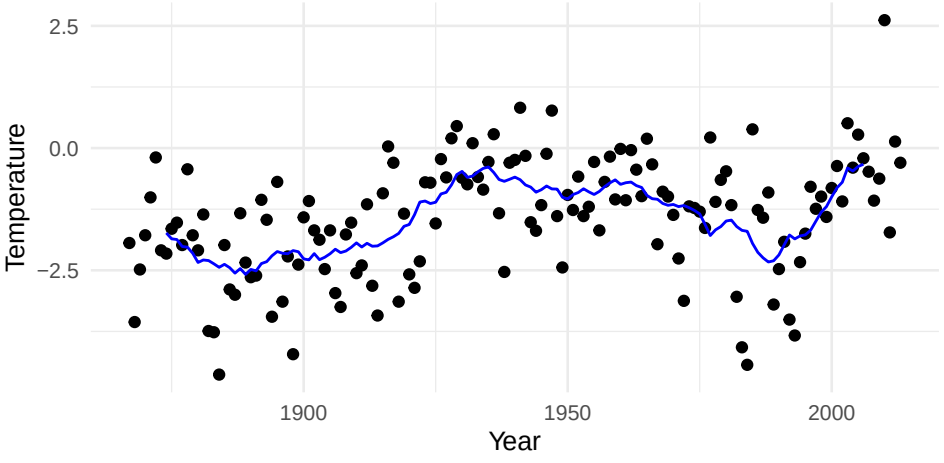


Figure 4: Nearest neighbor smoother with LOOCV-optimal $k = 15$.

Wiggliness

The running mean is wiggly!

We can use kernel smoothing (as we did last week) to fix this.

But another idea is to use smoothing splines instead.

Smoothing Splines

Smoothing Splines

We want a smooth function f that balances goodness-of-fit and wiggliness.

To do this, we minimize an objective function that penalizes wiggliness:

$$L(f) = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \underbrace{\int f''(z)^2 \mathrm{d}z}_{\|f''\|_2^2}$$

where $\lambda \geq 0$ is a tuning parameter that controls the trade-off.

The minimizer of this objective is a **cubic spline** with **knots** in the data points x_i.

That is, a function

$$f = \sum_j \beta_j \varphi_j$$

where φ_j is a basis function for the n-dimensional space of such splines.

The problem reduces to finding the coefficients $\boldsymbol{\beta}$.

In vector notation

$$\boldsymbol{f} = \boldsymbol{\Phi}\boldsymbol{\beta}$$

with $\boldsymbol{\Phi}_{ij} = \varphi_j(x_i)$, and

$$\begin{aligned} L(\boldsymbol{f}) &= (\boldsymbol{y} - \boldsymbol{f})^T(\boldsymbol{y} - \boldsymbol{f}) + \lambda \|f''\|_2^2 \\ &= (\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{\beta}) + \lambda \beta^T \boldsymbol{\Omega}\boldsymbol{\beta} \end{aligned}$$

with

$$\boldsymbol{\Omega}_{jk} = \int \varphi_j''(z)\varphi_k''(z)\mathrm{d}z.$$

Solution
The solution is a *linear smoother* with smoother matrix $\boldsymbol{S}_\lambda$ and penalty matrix $\boldsymbol{\Omega}$.

```
knots <- c(0, 0, 0, seq(0, 1, 0.2), 1, 1, 1)
xx <- seq(0, 1, 0.005)
b_splines <- splines::splineDesign(knots, xx)
```

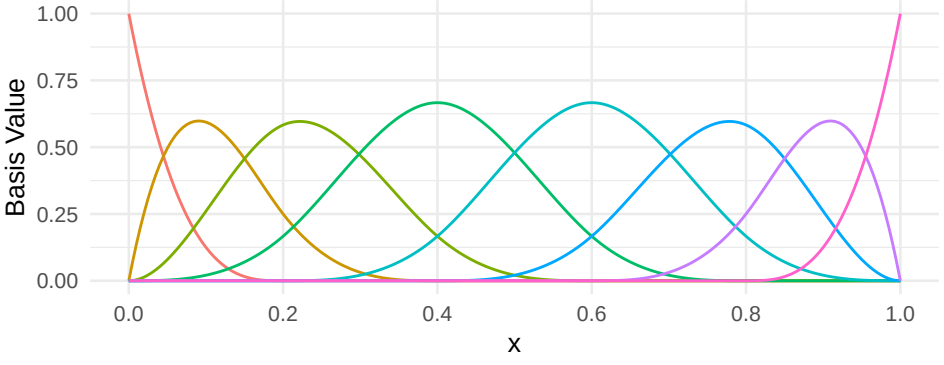


Figure 5: Cubic B-spline basis functions with knots at 0, 0.2, 0.4, 0.6, 0.8, and 1.

Penalty Matrix Ω

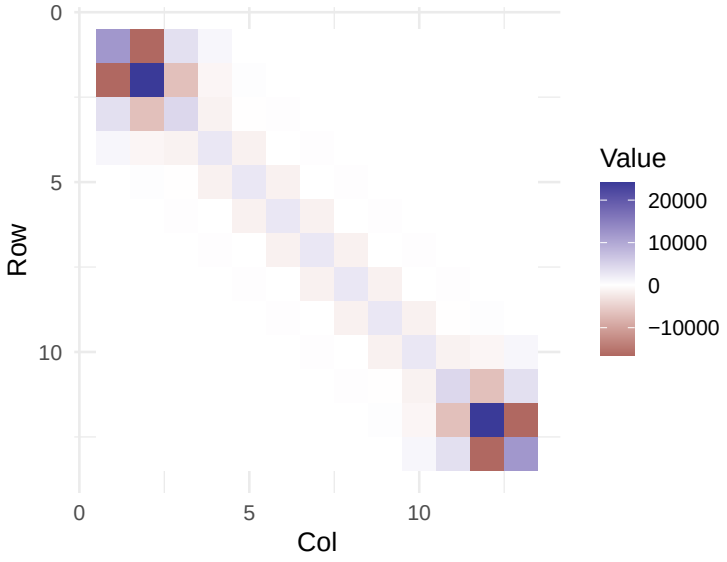


Figure 6: The penalty matrix Ω for a cubic spline with knots at 0, 0.2, ..., 0.8, 1.

Fitting a Smoothing Spline

We implement the matrix-algebra directly for computing $S_\lambda y$.

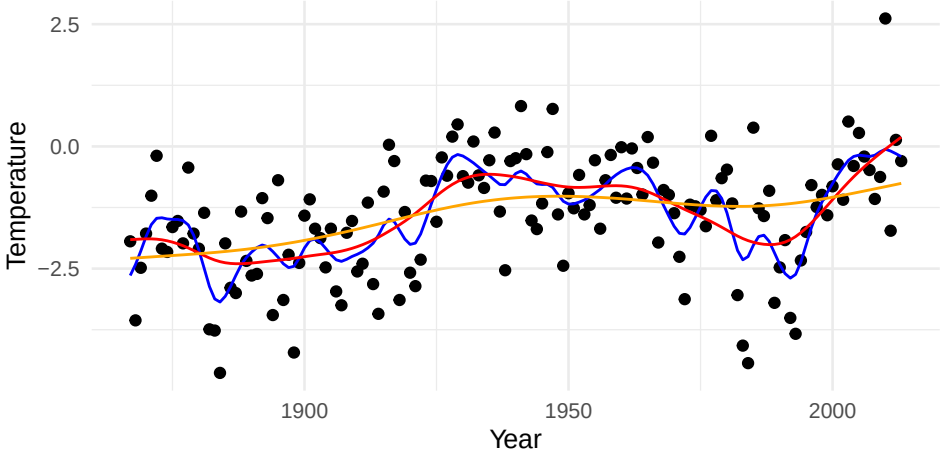
```
inner_knots <- nuuk_year$Year

# Note that order does not matter
knots <- c(rep(range(inner_knots), 3), inner_knots)

Phi <- splines::splineDesign(knots, inner_knots)
```

```
omega <- pen_mat(inner_knots) # Complicated, but depends only on knots
smoother <- function(lambda) {
  Phi %*%
    solve(
      crossprod(Phi) + lambda * omega, # Phi^T Phi + lambda Omega
      crossprod(Phi, nuuk_year$Temperature) # Phi^T y
    )
}
```

```
nuuk_plot +
  geom_line(aes(y = smoother(10)), color = "blue") +
  geom_line(aes(y = smoother(1000)), color = "red") +
  geom_line(aes(y = smoother(100000)), color = "orange")
```



Generalized Cross-Validation (GCV)

For smoothing splines, S_{ii} varies with i, which makes LOOCV expensive to compute.

Idea
Replace S_{ii} in LOOCV by df/n, with $\mathrm{df} = \mathrm{trace}(\boldsymbol{S}) = \sum_{i=1}^{n} S_{ii}$, to get the **generalized** cross-validation criterion

$$\mathrm{GCV} = \sum_{i=1}^{n} \left(\frac{y_i - \hat{f}_i}{1 - \mathrm{df}/n} \right)^2 .$$

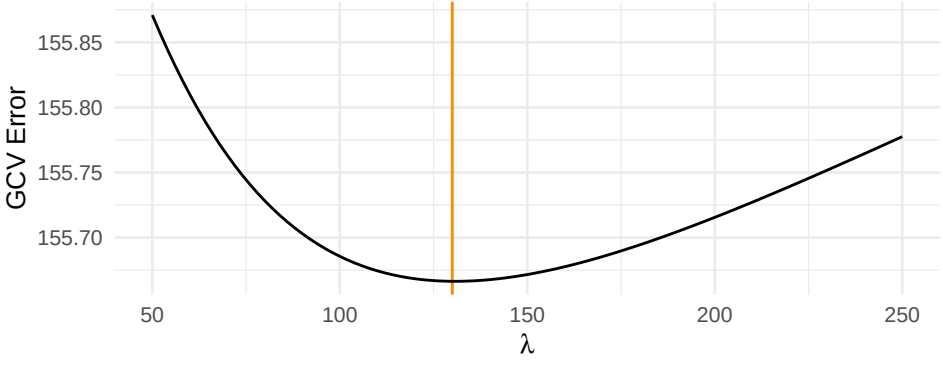
Implementation

```
gcv <- function(lambda, y) {
  S <- Phi %*% solve(crossprod(Phi) + lambda * omega, t(Phi))
  df <- sum(diag(S))
  sum(((y - S %*% y) / (1 - df / length(y)))^2, na.rm = TRUE)
}
```

GCV-Optimal λ

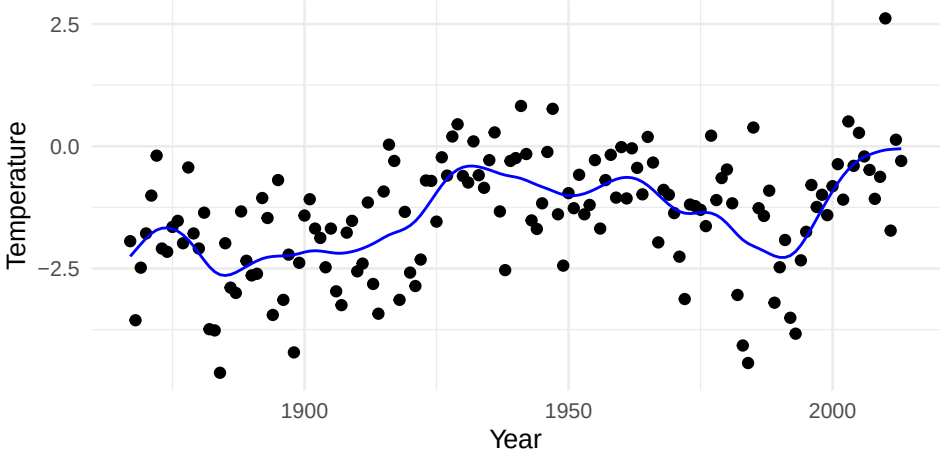
Apply `gcv()` across grid of λ-values and choose λ that minimizes GCV.

```
lambda <- seq(50, 250, 2)
gcv <- sapply(lambda, gcv, y = nuuk_year$Temperature)
lambda_opt <- lambda[which.min(gcv)]
```



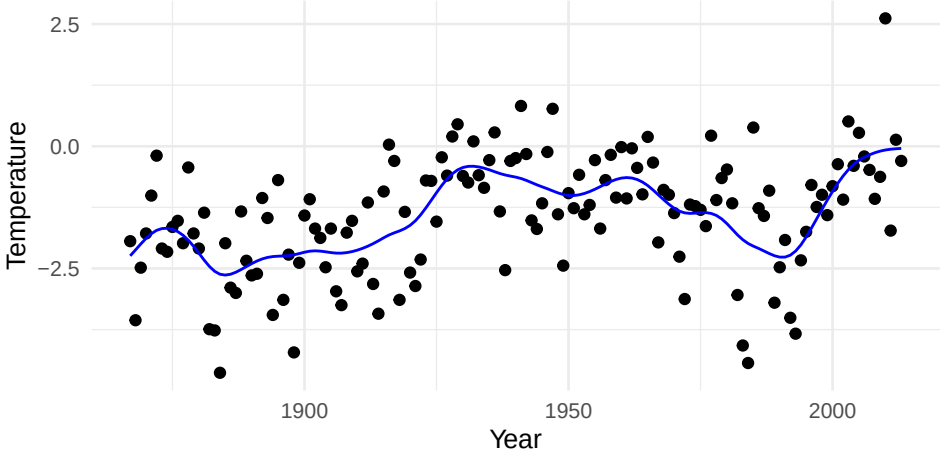
GCV-Optimal Smoothing Spline

```
smooth_opt <- smoother(lambda_opt)
nuuk_plot + geom_line(aes(y = smooth_opt), color = "blue")
```



Using smooth.spline()

```
smooth <- smooth.spline(nuuk_year$Year, nuuk_year$Temperature)
nuuk_plot + geom_line(aes(y = smooth$y), color = "blue")
```



Efficient Computations

When n is large, computing $\boldsymbol{S}_\lambda \boldsymbol{y}$ directly is expensive.

In practice, we therefore often use fewer basis functions than data points, $p < n$.

Singular Value Decomposition

Using the singular value decomposition

$$\boldsymbol{\Phi} = \boldsymbol{U} D \boldsymbol{V}^T$$

we can rewrite the smoother matrix as

$$\boldsymbol{S}_\lambda = \widetilde{\boldsymbol{U}}(I + \lambda \Gamma)^{-1} \widetilde{\boldsymbol{U}}^T$$

where $\widetilde{\boldsymbol{U}} = \boldsymbol{U}\boldsymbol{W}$ and

$$D^{-1}\boldsymbol{V}^T \boldsymbol{\Omega} \boldsymbol{V} D^{-1} = \boldsymbol{W} \Gamma \boldsymbol{W}^T.$$

The columns of $\widetilde{\boldsymbol{U}}$ form the **Demmler-Reinsch basis**.

Interpretation

Project the data y onto the Demmler-Reinsch basis (columns of $\widetilde{\boldsymbol{U}}$) to get coefficients $\hat{\boldsymbol{\beta}} = \widetilde{\boldsymbol{U}}^T y$.

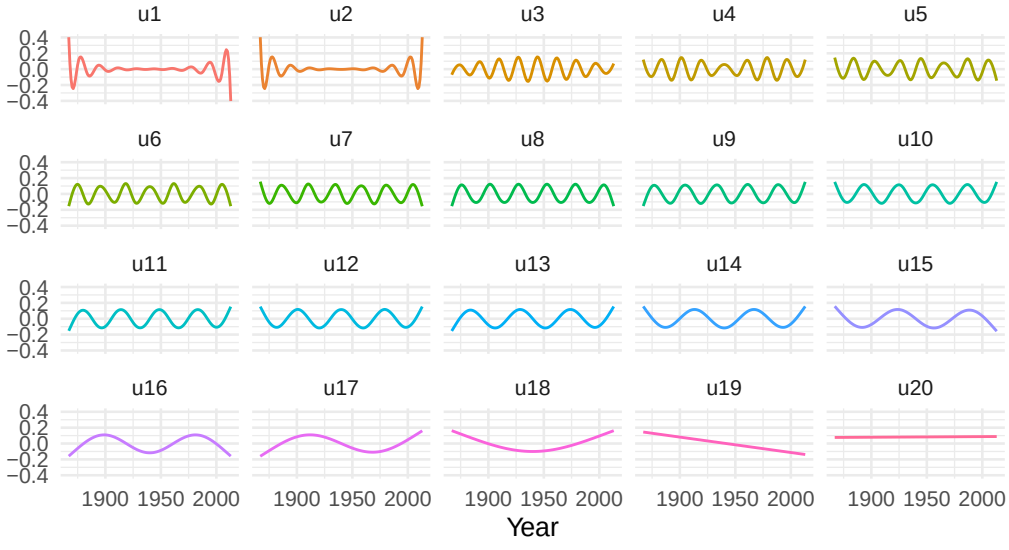
Each coefficient is shrunk according to the corresponding eigenvalue γ_i and smoothing parameter λ:

$$\hat{\boldsymbol{\beta}}_i(\lambda) = \frac{\hat{\beta}_i}{1 + \lambda\gamma_i}.$$

The final smoothed values are reconstructed by combining the shrunk coefficients with the basis functions.

$$\hat{\boldsymbol{f}} = \widetilde{\boldsymbol{U}}\hat{\boldsymbol{\beta}}(\lambda)$$

The Demmler-Reinsch Basis (Columns of $\widetilde{U}$)



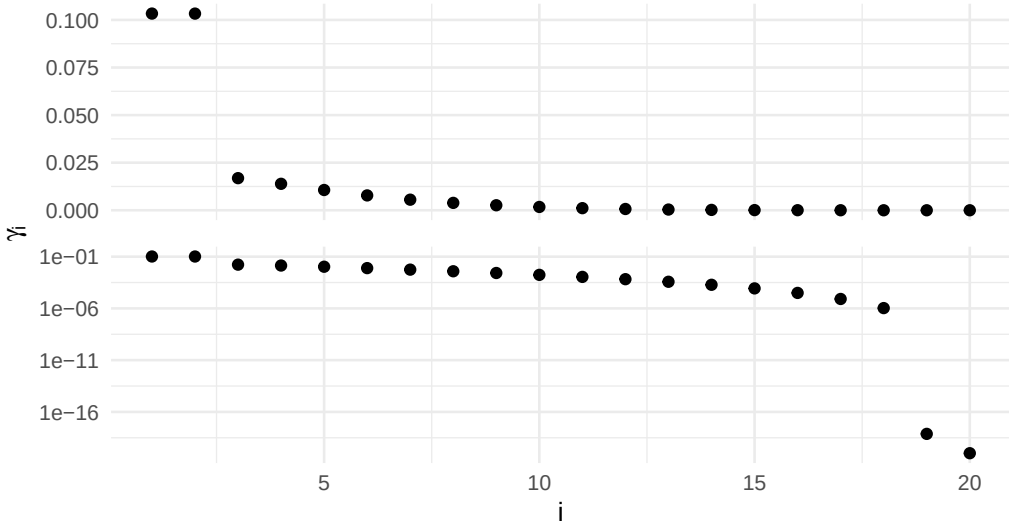


Figure 7: The eigenvalues γ_i of the penalty matrix in the Demmler-Reinsch basis.

Exercises

Exercises

Exercise 1: Parallel Square

Use the `foreach` and `doParallel` packages to compute the squares of a vector in parallel. Benchmark the parallel version against a regular `for` loop using `bench::mark()`.

Exercise 2: Future

Create a future that waits for 5 seconds using `Sys.sleep(5)`. When it's done, it should prints "The future is here!" and return the value 42. Try accessing the value of the future before and after the computation is complete.

Exercise 3: Smoothing Air Quality Data

Apply `stats::filter()` and `smooth.spline()` to the `airquality$Temp` data. Try different window sizes for the moving average and different `spar` values for the spline. Plot and compare the results.