Interview Guide

Research Scientist Interview Guide - 2026 Edition

Kundan Kumar

2026-05-10

Table of contents

Preface

"Think mathematically, reason scientifically, and build efficiently."

This is a structured preparation curriculum for Research Scientist roles at industrial AI labs, academic research groups, and applied ML organizations. It is built on a single conviction: every algorithm evolved to solve a pain point in its predecessor, and understanding that lineage is what separates a candidate who can recall methods from one who can reason about them.

The guide balances mathematical theory, statistical rigor, and hands-on implementation — tracing the evolution from classical machine learning through modern large language models and deep reinforcement learning systems.

The Lineage

Every method in this guide is positioned within an evolutionary chain:

```
Linear Models → Nonlinear Trees → Ensemble Boosting → Neural Networks
    → Transformers → LLMs → DRL → Physics-Informed & Federated Systems
```

Each chapter answers: *why was the next method needed, and what problem did it solve that its predecessor could not?*

Learning Philosophy

1. **Start from intuition.** Understand why each model exists before touching the math.
2. **Derive manually.** Write gradients, losses, and activations by hand before running code.
3. **Implement from scratch.** Rebuild models in PyTorch — no black boxes for core concepts.
4. **Visualize.** Use loss landscapes, weight evolution plots, and activation maps to build spatial intuition.
5. **Communicate.** Summarize each section as if presenting to a research committee.

Curriculum Overview

Module	Theme	Core Topics
1	Mathematical Foundations	Linear algebra, probability, optimization, gradient calculus, statistical testing
2	Algorithms & Data Engineering	Data structures, algorithm patterns, ML data pipelines
3	Classical Machine Learning	Regression, SVM, trees, ensembles, dimensionality reduction, clustering, time series
4	Deep Learning	Perceptrons, CNNs, RNNs, Transformers, computer vision
5	Deep Reinforcement Learning	Q-learning, DDPG, PPO, SAC, GRPO, constrained RL, RLHF
6	Large Language Models	Transformer theory, attention, fine-tuning, LoRA, RLHF, alignment
7	Model Compression & Deployment	Pruning, quantization, distillation, inference optimization, scalability
8	Research Design & Evaluation	Ablations, reproducibility, model evaluation, uncertainty quantification
9	Trustworthy & Robust AI	Adversarial robustness, fairness, differential privacy, alignment
10	Emerging Topics	Diffusion models, neural operators, causal learning, multimodal systems
A	Appendices	Derivations, regression extensions, clustering, practice problems

---

Interview Preparation Framework

Phase	Weeks	Focus	Key Deliverable
Phase 1 — Core Concepts	1–4	ML/DL fundamentals, mathematical review	Linear models and neural nets from scratch
Phase 2 — Deep RL	5–7	PPO, DDPG, constrained RL	End-to-end DRL implementation with ablation
Phase 3 — LLMs	8–10	Pretraining, fine-tuning, LoRA, RLHF	GPT-2 from scratch + LoRA fine-tuning

Phase	Weeks	Focus	Key Deliverable
Phase 4 — Research Communication	11	Technical writing, experiment narration	Short technical blog or mini-paper
Phase 5 — Systems & Scalability	12–13	Compression, quantization, deployment	Inference profiling report
Phase 6 — Mock Interviews I	14	Technical + research + system design	Self-evaluation rubric
Phase 7 — Mock Interviews II	15	Full simulation under time constraint	Final preparation checklist

---

Prerequisites

Readers are expected to have foundational knowledge in:

- **Mathematics** — linear algebra, multivariate calculus, probability theory
- **Programming** — Python, PyTorch or TensorFlow, basic data manipulation
- **Statistics** — hypothesis testing, regression, inference
- **CS Fundamentals** — data structures, algorithms, complexity analysis

---

Author

Kundan Kumar — https://kundan-kumarr.github.io/

Longer notes and weekly write-ups: kundan-kumarr.github.io/blog · neuravp.substack.com

Citation

Kumar, Kundan. (`r substr(Sys.Date(), 1, 4)`). *From First Principles — Research Scientist Interview Guide*. `r substr(Sys.Date(), 1, 4)` Edition.

License

Part I.

Module 1 - Mathematical Foundations

1. Statistics & Mathematics

After this lecture, you should be competent (again) in assessing violations of various assumptions of linear regression models, particularly assumptions about model residuals, by being able to apply visual assessments and formal statistical tests, and interpret the results and effects of the violations.

Objectives

1. Recall the form and standard assumptions of linear regression models.
2. Recall and apply standard methods of assessing and testing homogeneity of variance and normality of residuals.
3. Define and test independence (most often, uncorrelatedness) of regression model residuals.

Reading materials

- Chapters 3–4 in Chatterjee and Hadi (2006)

1.1. Diagnostics for the simple linear regression: residual analysis

Given a simple linear regression (SLR) model

$$Y_t = \beta_0 + \beta_1 X_t + \epsilon_t,$$

where Y_t is the dependent variable and X_t is the regressor (independent, predictor) variable, $t = 1, \dots, n$, and n is the sample size.

The Gauss–Markov theorem

If ϵ_t are uncorrelated random variables with common variance, then of all possible estimators β_0^* and β_1^* that are linear functions of Y_t, the least squares estimators have the smallest variance.

Thus, the ordinary least squares (OLS) assumptions are:

1. the residuals ϵ_t have common variance (ϵ_t are homoskedastic);
2. the residuals ϵ_t are uncorrelated;

   to provide prediction intervals (PIs), confidence intervals (CIs), and to test hypotheses about the parameters in our model, we also need to assume that
3. the residuals ϵ_t are normally distributed ($\epsilon_t \sim N(0, \sigma^2)$).

> **Note**
>
> If the residuals are independent and identically distributed and normal ($\epsilon_t \sim$ i.i.d. $N(0, \sigma^2)$), then all three above properties are automatically satisfied. In this case, ϵ_t are not only

uncorrelated but are independent. To be independent is a much stronger property than to be uncorrelated.

Note

While a given model may still have useful predictive value even when the OLS assumptions are violated, the confidence intervals, prediction intervals, and p-values associated with the t-statistics will generally be incorrect when the OLS assumptions do not hold.

A basic technique for investigating the aptness of a regression model is based on analyzing the residuals ϵ_t. In a residual analysis, we attempt to assess the validity of the OLS assumptions by examining the estimated residuals $\hat{\epsilon}_1, \ldots, \hat{\epsilon}_n$ to see if they satisfy the imposed conditions. If the model is apt, the observed residuals should reflect the assumptions listed above.

We perform our diagnostics analysis from a step-by-step verification of each assumption. We start with visual diagnostics, then proceed with formal tests. A lot of useful diagnostic information may be obtained from a residual plot.

1.1.1. Homoskedasticity

We plot the residuals $\hat{\epsilon}_t$ vs. time, fitted values $\hat{Y}_t$, and predictor values X_t. If the assumption of constant variance is satisfied, $\hat{\epsilon}_t$ fluctuate around the zero mean with more or less constant amplitude and this amplitude does not change with time, fitted values $\hat{Y}_t$, and predictor values X_t.

If the (linear) model is not appropriate, the mean of the residuals may be non-constant, i.e., not always 0. Figure 1.1 shows an example of a random pattern that we would like the residuals to have (no systematic patterns).

```
set.seed(1)
n = 26; m = 0; s = 522
x <- ts(rnorm(n, mean = m, sd = s))
forecast::autoplot(x) +
    geom_hline(yintercept = 0, lty = 2, col = 4)
```

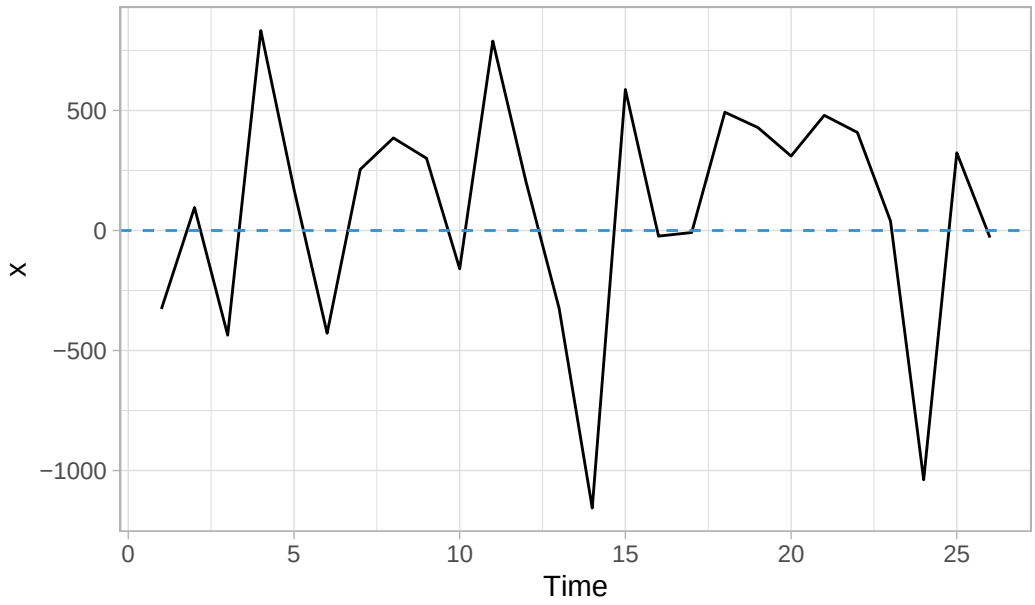


Figure 1.1.: A time series plot of 'ideal' residuals. These residuals x_t are simulated i.i.d. normal.

What can we notice in a residual plot?

- Change of variability with time indicates heterogeneity of variance of the residuals.
- Obvious lack of symmetry (around 0) in the plot suggests a lack of normality or presence of outliers.
- Systematic trends in the residuals suggest correlations between the residuals or inadequateness of the proposed model.

Sometimes it is possible to transform the dependent or independent variables to remedy these problems, i.e., to get rid of the correlated residuals or to stabilize the variance (see Appendix A and Appendix B). Otherwise, we need to change (re-specify) the model.

A useful technique that can guide us in this process is to plot $\hat{\epsilon}_t$ vs. $\hat{Y}_t$ and $\hat{\epsilon}_t$ vs. each predictor X_t. Similarly to their time series plot, $\hat{\epsilon}_t$ should fluctuate around the zero mean with more or less constant amplitude.

Example: Dishwasher shipments model and patterns in residuals

Figure 1.2 shows the R code and residuals of a simple linear regression exploring dishwasher shipments (DISH) and private residential investments (RES) for several years.
How different are the patterns in Figure 1.2 from those in Figure 1.1?

```
D <- read.delim("data/dish.txt") %>%
    rename(Year = YEAR)
mod1 <- lm(DISH ~ RES, data = D)
p1 <- ggplot(D, aes(x = Year, y = mod1$residuals)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    ylab("Residuals")
p2 <- ggplot(D, aes(x = mod1$fitted.values, y = mod1$residuals)) +
    geom_point() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    xlab("Fitted values") +
    ylab("Residuals")
p3 <- ggplot(D, aes(x = RES, y = mod1$residuals)) +
    geom_point() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    xlab("Residential investments (RES)") +
    ylab("Residuals")
p1 + p2 + p3 +
    plot_annotation(tag_levels = 'A')
```

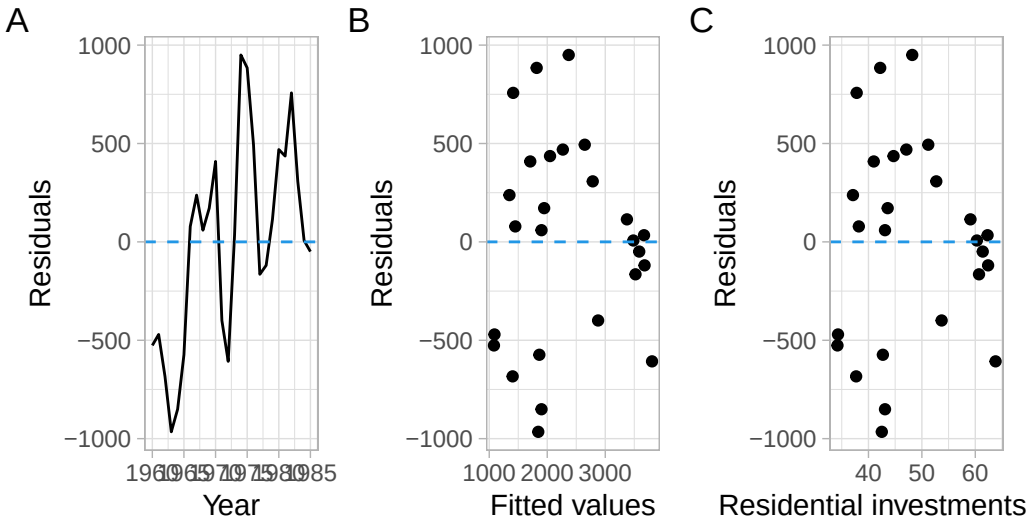


Figure 1.2.: Estimated residuals plotted vs. time, fitted values, and predictor.

In Figure 1.2, we see a pattern of residuals increasing in time, and a pattern of lower variability for high fitted values or high residential investments. Thus, the assumption of homoskedasticity is violated. In Figure 1.1, no such patterns are observed.

1.1.2. Uncorrelatedness

It is a deep topic that we shall discuss many times in different variations in the future. When observations are obtained in a time sequence (the topic of time series analysis and our course), there is a *high* possibility that the errors ϵ_t are correlated. For instance, if the residual is positive (or negative) for a given day t, it is likely that the residual for the following day $t+1$ is also positive (or negative). Such residuals are said to be *autocorrelated* (i.e., serially correlated). Autocorrelation of many environmental time series is positive.

When the residuals ϵ_t are related over time, a model for the residuals frequently employed is the first-order autoregressive model, i.e., the AR(1) model.

The *autoregressive model of the first order*, AR(1), is defined as

$$\epsilon_t = \rho\epsilon_{t-1} + u_t,$$

where ρ is the autoregression coefficient ($-1 < \rho < 1$) and u_t is an uncorrelated $N(0, \sigma^2)$ time series.

The model assumes that the residual ϵ_t at the time t contains a component resulting from the residual ϵ_{t-1} at the time $t-1$ and a random disturbance u_t that is independent of the earlier periods.

Effects of autocorrelation

If the OLS method is employed for the parameter estimation and the residuals ϵ_t are autocorrelated of the first order, then the consequences are:

- The OLS estimators will still be unbiased, but they no longer have the minimum variance property (see the Gauss–Markov theorem); they tend to be relatively inefficient.
- The residual mean square error (MSE) can seriously underestimate the true variance of the error terms in the model.
- Standard procedures for CI, PI, and tests using the F and Student's t distributions are no longer strictly applicable.

For example, see Section 5.2 in Chatterjee and Simonoff (2013) for more details.

Durbin–Watson test

A widely used test for examining whether the residuals in a regression model are correlated is the Durbin–Watson test. This test is based on the AR(1) model for ϵ_t. The one-tail test alternatives are

$$H_0\colon \rho = 0 \quad vs. \quad H_1\colon \rho > 0, \tag{1.1}$$
$$H_0\colon \rho = 0 \quad vs. \quad H_1\colon \rho < 0, \tag{1.2}$$

and the two-tail test is

$$H_0\colon \rho = 0 \quad vs. \quad H_1\colon \rho \neq 0.$$

> **Note**
>
> When dealing with real data, positive autocorrelation is usually the case.

The Durbin–Watson test statistic DW is based on the differences between the adjacent residuals, $\epsilon_t - \epsilon_{t-1}$, and is of the following form:

$$\mathrm{DW} = \frac{\sum_{t=2}^{n} (\epsilon_t - \epsilon_{t-1})^2}{\sum_{t=1}^{n} \epsilon_t^2},$$

where ϵ_t is the regression residual at the time t and n is the number of observations.

The DW statistic takes on values in the range $[0, 4]$. In fact,

- When ϵ_t are positively correlated, adjacent residuals tend to be of similar magnitude so that the numerator of DW will be relatively small or 0.
- When ϵ_t are negatively correlated, adjacent residuals tend to be of similar magnitude but with the opposite sign so that the numerator of DW will be relatively large or equal to 4.

Hence, low DW corresponds to positive autocorrelation. Values of DW that tend towards 4 are in the region for negative autocorrelation.

The exact action limit for the Durbin–Watson test is difficult to calculate. Hence, the test is used with a lower bound d_L and an upper bound d_U. We may use Table 1.1 as a rule of thumb.

Table 1.1.: Regions of rejection of the null hypothesis for the Durbin–Watson test

from 0 to d_L	from d_L to d_U	from d_U to $4-d_U$	from $4-d_U$ to $4-d_L$	from $4-d_L$ to 4
Reject H_0, positive autocorrelation	Neither accept H_1 or reject H_0	Do not reject H_0	Neither accept H_1 or reject H_0	Reject H_0, negative autocorrelation

The critical values d_L and d_U have been tabulated for combinations of various sample sizes, significance levels, and number of regressors in a model. For large samples, a normal approximation can be used (Chatterjee and Simonoff 2013):

$$z = \left(\frac{\mathrm{DW}}{2} - 1\right)\sqrt{n}.$$

Statistical software packages usually provide exact p-values based on the null distribution of the test statistic (a linear combination of χ^2 variables).

Example: Dishwasher residuals DW test

Apply the Durbin–Watson test to the residuals from the dishwashers example, i.e., `DISH` vs. `RES`, using the R package `lmtest`.

```
lmtest::dwtest(D$DISH ~ D$RES, alternative = "greater")
```

```
#>
#>  Durbin-Watson test
#>
#> data:  D$DISH ~ D$RES
#> DW = 0.6, p-value = 3e-06
```

```
#> alternative hypothesis: true autocorrelation is greater than 0
```

Based on the low p-value we can reject the H_0: $\rho = 0$ at the 95% confidence level and accept the alternative H_1: $\rho > 0$.

Runs test

Departures of randomness can take so many forms that no single test for randomness is best for all situations. For instance, one of the most common departures from randomness is the tendency of a sequence to persist in its direction of movement.

We can count the number of times a sequence of observations crossed a cut-off line, for example, the median line, and use this information to assess the randomness of ϵ_t. Alternatively, we count successions of positive or negative differences (see Section 1.4 on the difference sign test). Each such succession is called a *run*.

The formal test is the following. When a sequence of N observations with n observations in positive runs and m observations in negative runs is a random process with independent values generated from a continuous distribution, then the sampling distribution of the number of runs R has the mean and variance

$$\mathrm{E}(R) = \frac{1 + 2nm}{N}, \qquad \sigma^2(R) = \frac{2nm(2nm - n - m)}{N^2(N-1)},$$

where $N = n + m$ is the total sample size.

The only assumption for this test is that all sample observations come from a continuous distribution.

The two-tail alternative is as follows

- H_0: Sequence is generated by a random process;
- H_1: Sequence is generated by a process containing either persistence or frequent changes in direction.

When positive autocorrelation (or persistence) is present, R will be small. On the other hand, if the process involves frequent changes in direction (negative autocorrelation or anti-persistence), R will be too large.

When the number of observations is sufficiently large, i.e., $N > 30$, the runs test statistic R is based on the standardized normal test statistic

$$z = \frac{R - \mathrm{E}(R)}{\sigma(R)}.$$

Here z follows approximately a standard normal distribution.

Runs test is easy to interpret. Runs test allows assessing only the first-order serial correlation in the residuals, i.e., to test whether two residuals that are one lag apart are correlated.

Example: Dishwasher residuals runs test

```
par(mfrow = c(1, 2))
lawstat::runs.test(x, plot.it = TRUE)
```

```
#>
#>  Runs Test - Two sided
#>
#> data:  x
#> Standardized Runs Statistic = -0.4, p-value = 0.7
```

```
lawstat::runs.test(mod1$residuals, plot.it = TRUE)
```

```
#>
#>  Runs Test - Two sided
#>
#> data:  mod1$residuals
#> Standardized Runs Statistic = -3, p-value = 0.005
```

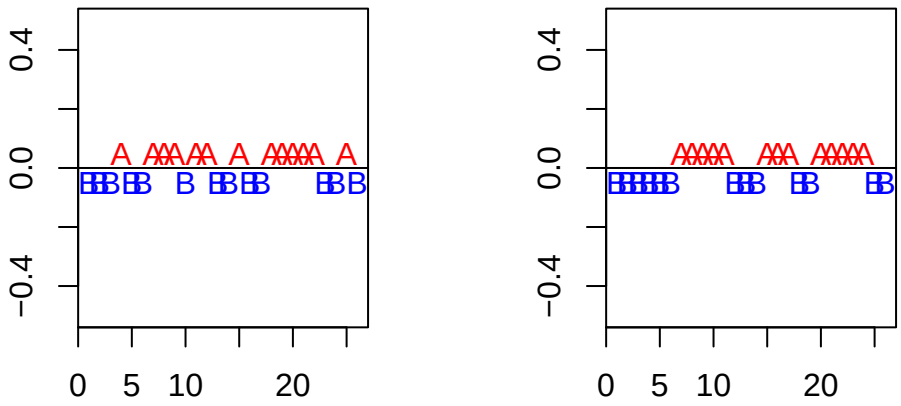


Figure 1.3.: Runs tests and plots of independent normally distributed simulations x_t and the `DISH` residuals.

The p-value for the runs test for the residuals is very low (Figure 1.3), which supports the findings of the DW test that residuals are first-order serially correlated.

1.1.3. Normality

There are two major ways of checking normality. Graphical methods visualize differences between empirical data and theoretical normal distribution. Numerical methods conduct statistical tests on the null hypothesis that the variable is normally distributed.

Graphical methods

Graphical methods visualize the data using graphs, such as histograms, stem-and-leaf plots, box plots, etc. For example, Figure 1.4 shows a histogram of the simulated normally distributed data and the residuals from the dishwasher example with superimposed normal curves with the corresponding mean and standard deviation.

```
p1 <- ggplot(data.frame(x = x), aes(x = x)) +
    geom_histogram(aes(y = after_stat(density)), binwidth = 300, fill = "grey50") +
    stat_function(fun = dnorm,
                  args = list(mean = mean(x), sd = sd(x)),
                  col = 1, lwd = 1.5) +
    ylab("Density") +
    ggtitle("Random normal values")
p2 <- ggplot(x, aes(x = mod1$residuals)) +
    geom_histogram(aes(y = after_stat(density)), binwidth = 300, fill = "grey50") +
    stat_function(fun = dnorm,
                  args = list(mean = mean(mod1$residuals), sd = sd(mod1$residuals)),
                  col = 1, lwd = 1.5) +
    ylab("Density") +
    ggtitle("Model residuals")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

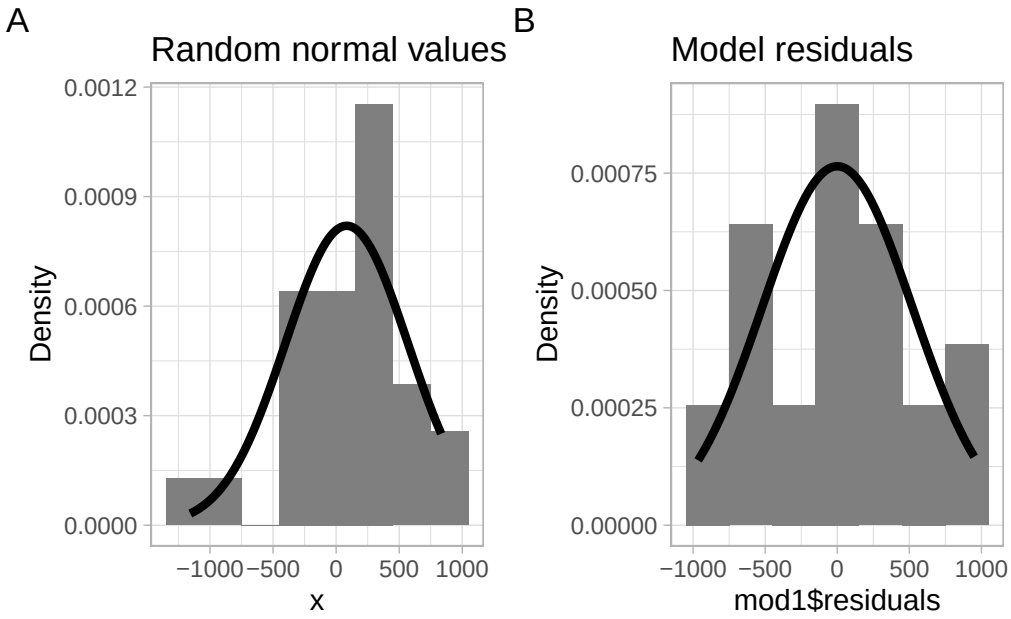


Figure 1.4.: Histograms of the simulated normally distributed values and estimated regression residuals.

Another very popular graphical method of assessing normality is the quantile-quantile (Q-Q) plot. The Q-Q plot compares the ordered values of a variable with the corresponding ordered values of the normal distribution.

> **Note**
>
> Q-Q plots can also be used to compare sample quantiles with quantiles of other, not normal, distribution (e.g., t or gamma distribution), or to compare quantiles of two samples (to assess if both samples come from the same, unspecified, distribution).

Let X be a random variable having the property that the equation

$$\Pr(X \leqslant x) = \alpha$$

has a unique solution $x = x_{(\alpha)}$ for each $0 < \alpha < 1$. That is, there exists $x_{(\alpha)}$ such that

$$\Pr\left(X \leqslant x_{(\alpha)}\right) = \alpha \tag{1.3}$$

and no other value of x satisfies Equation 1.3. Then we will call $x_{(\alpha)}$ the αth *(population) quantile* of X. Note that any normal distribution has this uniqueness property. If we consider a standard normal $Z \sim N(0, 1)$, then some well-known quantiles are:

- $z_{(0.5)} = 0$ (the median), `qnorm(0.5, mean = 0, sd = 1)`
- $z_{(0.05)} = -1.645$ and $z_{(0.95)} = 1.645$
- $z_{(0.025)} = -1.96$ and $z_{(0.975)} = 1.96$

We call the 0.25th, 0.5th, 0.75th quantiles the first, the second, and the third quartiles, respectively. The quartiles divide our data into 4 equal parts.

Now suppose $X \sim N(\mu, \sigma^2)$. By standardizing to $Z \sim N(0,1)$, we obtain

$$\alpha = \Pr\left(X \leqslant x_{(\alpha)}\right) = \Pr\left(\frac{X-\mu}{\sigma} \leqslant \frac{x_{(\alpha)}-\mu}{\sigma}\right) = \Pr\left(Z \leqslant \frac{x_{(\alpha)}-\mu}{\sigma}\right).$$

We also have $\alpha = \Pr(Z \leqslant z_{(\alpha)})$ by definition. It follows that

$$z_{(\alpha)} = \frac{x_{(\alpha)}-\mu}{\sigma} \quad \text{and hence} \quad x_{(\alpha)} = \sigma z_{(\alpha)} + \mu.$$

Thus, if X is truly normal, a plot of the quantiles of X vs. the quantiles of the standard normal distribution should yield a straight line. A plot of the quantiles of X vs. the quantiles of Z is called a Q-Q plot.

Estimating quantiles from data

Let $X_1, \ldots, X_n$ be a sequence of observations. Ideally, $X_1, \ldots, X_n$ should represent i.i.d. observations but we will be happy if preliminary tests indicate that they are homoskedastic and uncorrelated (see the previous sections). We order them from the smallest to the largest and indicate this using the notation

$$X_{(1/n)} < X_{(2/n)} < X_{(3/n)} < \cdots < X_{((n-1)/n)} < X_{(n/n)}.$$

The above ordering assumes no ties, but ties can be quite common in data, even continuous data, because of rounding. As long as the proportion of ties is small, this method can be used.

Note that the proportion of observations less than or equal to $X_{(k/n)}$ is exactly k/n. Hence $X_{(k/n)}$, called the kth *sample quantile*, is an estimate of the population quantile $x_{(k/n)}$.

The normal Q-Q plot is obtained by plotting the sample quantiles vs. the quantiles of the standard normal distribution. The base-R function `qqnorm()` produces a normal Q-Q plot of data and the function `qqline()` adds a line that passes through the first and third quartiles. The R package `ggplot2` has analogous functions `ggplot2::stat_qq()` and `ggplot2::stat_qq_line()`. The R packages `car` and `ggpubr` also draw Q-Q plots, but also add a point-wise confidence envelope with their functions `car::qqPlot()` (base-R plot) and `ggpubr::ggqqplot()` (ggplot-type plot).

Example: Dishwasher residuals normal Q-Q plot

Figure 1.5 shows the Q-Q plots of the residuals from the dishwasher example and the simulated normal data with the same mean and standard deviation.

```
p1 <- ggpubr::ggqqplot(x) +
    ggtitle("Random normal values") +
    xlab("Standard normal quantiles")
p2 <- ggpubr::ggqqplot(mod1$residuals) +
    ggtitle("Model residuals") +
    xlab("Standard normal quantiles")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

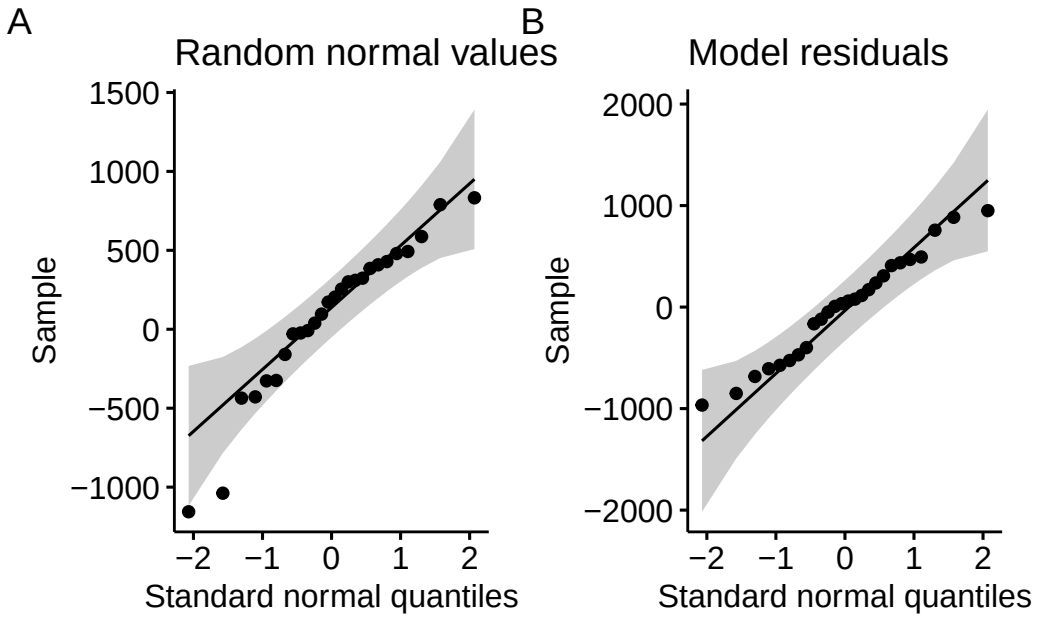


Figure 1.5.: Normal Q-Q plots of the normally distributed simulated values x_t and the dishwasher residuals.

Both Q-Q plots in Figure 1.5 show a good correspondence of the sample quantiles with theoretical normal quantiles, providing no sufficient evidence against normality of the underlying distributions.

Although visually appealing, these graphical methods do not provide objective criteria to determine the normality of variables.

Shapiro–Wilk normality test

One of the most popular numerical methods for assessing normality is the Shapiro–Wilk (SW) test:

- H_0: the sample data come from a normally distributed population;
- H_1: the population is not normally distributed).

The SW test is the ratio of the best estimator of the variance to the usual corrected sum of squares estimator of the variance. It has been originally constructed by considering the regression of ordered sample values on corresponding expected normal order statistics. The SW statistic is given by

$$\mathrm{SW} = \frac{\left(\sum a_i x_{(i)}\right)^2}{\sum \left(x_i - \bar{x}\right)^2},$$

where $x_{(i)}$ are the ordered sample values ($x_{(1)}$ is the smallest) and the a_i are constants generated from the means, variances, and covariances of the order statistics of a sample of size n from a normal distribution. The SW statistic lies between 0 and 1. If the SW statistic is close to 1, this indicates the normality of the data. The SW statistic requires the sample size n to be between 7 and 2000.

Example: Dishwasher residuals normality test

Based on the p-values below, we cannot reject the null hypothesis of normality in both cases.

```
shapiro.test(x)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  x
#> W = 0.9, p-value = 0.08
```

```
shapiro.test(mod1$residuals)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  mod1$residuals
#> W = 1, p-value = 0.8
```

1.1.4. Summary of the simple linear regression residual diagnostics

1. The residuals do not have a constant mean.
2. The residuals do not seem to have a constant variance.
3. The residuals are positively correlated.
4. The residuals look normally distributed (but the SW statistic might be affected by the serial correlation of the residuals).

1.2. Multiple linear regression

Here we consider a case of p explanatory variables

$$Y_t = \beta_0 + \beta_1 X_{t,1} + \cdots + \beta_p X_{t,p} + \epsilon_t \quad (t = 1, \ldots, n).$$

This can be expressed more compactly in a matrix notation as

$$Y = X\beta + \epsilon,$$

where $Y = (Y_1, \ldots, Y_n)^\top$, $\beta = (\beta_0, \ldots, \beta_p)^\top$, $\epsilon = (\epsilon_1, \ldots, \epsilon_n)^\top$; X is an $n \times (p+1)$ design matrix

$$X = \begin{pmatrix} 1 & X_{1,1} & \ldots & X_{1,p} \\ 1 & X_{2,1} & \ldots & X_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & X_{n,1} & \ldots & X_{n,p} \end{pmatrix}.$$

Here the historical dataset for the dependent variable consists of the observations $Y_1, \ldots, Y_n$; the historical dataset for the independent variables consists of the observations in the matrix X.

Minimizing $SSE = (Y - X\hat{\beta})^\top (Y - X\hat{\beta})$ yields the least squares solutions

$$\hat{\beta} = \left(X^\top X\right)^{-1} X^\top Y$$

for non-singular $X^\top X$.

The forecast of a future value Y_t is then given by

$$\hat{Y}_t = x_t^\top \hat{\beta},$$

where x_t is a (column) vector at the time t.

Under the OLS assumptions (recall them), we obtain

$$\text{var}\left(\hat{\beta}_j\right) = \sigma^2 \left(X^\top X\right)^{-1}_{jj},$$

where the $\left(X^\top X\right)^{-1}_{jj}$ denotes the jth diagonal element of $\left(X^\top X\right)^{-1}$.

This yields

$$s.e.\left(\hat{\beta}_j\right) = \hat{\sigma}\sqrt{\left(X^\top X\right)^{-1}_{jj}}.$$

Note that here the degrees of freedom (d.f.) are $n - (p+1) = n - p - 1$. (The number of estimated parameters for the independent variables is p, plus one for the intercept, i.e., $p + 1$.)

Under the OLS assumptions, a $100(1-\alpha)\%$ confidence interval for the parameter β_j $(j = 0, 1, \ldots, p)$ is given by

$$\begin{aligned} &\hat{\beta}_j \pm t_{\alpha/2, n-(p+1)} s.e.\left(\hat{\beta}_j\right) \text{ or} \\ &\hat{\beta}_j \pm t_{\alpha/2, n-(p+1)} \hat{\sigma}\sqrt{\left(X^\top X\right)^{-1}_{jj}}. \end{aligned} \tag{1.4}$$

Typically, $s.e.(\hat{\beta}_j)$ is available directly from the R output, so Equation 1.4 is calculated automatically.

Under the OLS assumptions, it can be shown that

$$\text{var}\left(Y_t - \hat{Y}_t\right) = \sigma^2 \left(x_t^\top \left(X^\top X\right)^{-1} x_t + 1\right),$$

yielding a $100(1-\alpha)\%$ prediction interval for Y_t:

$$x_t^\top \hat{\beta} \pm t_{\alpha/2, n-(p+1)} \hat{\sigma}\sqrt{x_t^\top \left(X^\top X\right)^{-1} x_t + 1}.$$

We usually never perform these calculations by hand and will use the corresponding software functions, e.g., using the function `predict()`, see an example code below.

What else can we get from the regression output?

As in SLR, we will look for the t-statistics and p-values to get an idea about the statistical significance of each of the predictors $X_{t,1}, X_{t,2}, \ldots, X_{t,p}$. The confidence intervals constructed above correspond to individual tests of hypothesis about a parameter, i.e., H_0: $\beta_j = 0$ vs. H_1: $\beta_j \neq 0$.

We can also make use of the F-test. The F-test considers *all* parameters (other than the intercept β_0) simultaneously, testing

$$\begin{aligned} &H_0\colon \beta_1 = \cdots = \beta_p = 0 \quad \text{vs.} \\ &H_1\colon \beta_j \neq 0 \quad \text{for at least one} \quad j \in \{1, \ldots, p\}. \end{aligned}$$

Formally, $F_{\text{obs}} = \text{MSR}/\text{MSE}$ (the ratio of the mean square due to regression and the mean square due to stochastic errors).

We reject H_0 when F_{obs} is too large relative to a cut-off point determined by the degrees of freedom of the F-distribution. The p-value for this F-test is provided in the `lm()` output. Rejecting H_0 is equivalent to stating that the model has some explanatory value within the range of the data set, meaning that changes in at least some of the explanatory X-variables correlate to changes in the average value of Y.

Recall that

$$\begin{aligned} \text{SST} &= \sum_{i=1}^{n} (Y_t - \overline{Y})^2 = \text{SSR} + \text{SSE}, \\ \text{SSE} &= \sum_{t=1}^{n} (Y_t - \hat{\beta}_0 - \hat{\beta}_1 X_{t,1} - \cdots - \hat{\beta}_p X_{t,p}) \end{aligned}$$

and, hence,

$$\text{SSR} = \text{SST} - \text{SSE}.$$

To conclude that the model has a reasonable fit, however, we would additionally like to see a high R^2 value, where

$$R^2 = \text{SSR}/\text{SST}$$

is the proportion of the total sum of squares explained by the regression.

Small R^2 means that the stochastic fluctuations around the regression line (or prediction equation) are large, making the prediction task difficult, even though there may be a genuine explanatory relationship between the average value $\text{E}(Y)$ and some of the X-variables.

Another criterion to judge the aptness of the obtained model is the adjusted R^2:

$$R^2_{adj} = 1 - \frac{n-1}{n-p}\left(1 - R^2\right).$$

Unlike R^2 itself, R^2_{adj} need not increase if an arbitrary (even useless) predictor is added to the model because of the correction $(n-1)/(n-p)$.

> **Note**
>
> The intercept β_0 is not included in the F-test because there is no explanatory variable associated with it. In other words, β_0 does not contribute to the regression part of the model.

Example: Dishwasher shipments multiple linear regression

Let us now extend the SLR model that we considered previously and include another potential predictor, the durable goods expenditures (billion of 1972 dollars). The goal is to build a model to predict the unit factory shipments of dishwashers (DISH) vs. private residential investment (RES) and durable goods expenditures (DUR) using a multiple linear regression model (MLR):

$$Y_t = \beta_0 + \beta_1 X_{t,1} + \beta_2 X_{t,2} + \epsilon_t,$$

where $X_{t,1}$ is the private residential investment `RES`; $X_{t,2}$ is the durable goods expenditures `DUR`.
Now we apply the OLS method to estimate the coefficients β_0, β_1, and β_2 using the function `lm()`.

```
mod2 <- lm(DISH ~ RES + DUR, data = D)
summary(mod2)
```

```
#>
#> Call:
#> lm(formula = DISH ~ RES + DUR, data = D)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -643.9 -263.2  -22.1  190.2  920.0
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -1603.09     391.31   -4.10  0.00044 ***
#> RES            50.97      11.37    4.48  0.00017 ***
#> DUR            13.77       2.78    4.95  5.2e-05 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 378 on 23 degrees of freedom
#> Multiple R-squared:  0.877,  Adjusted R-squared:  0.867
#> F-statistic: 82.3 on 2 and 23 DF,  p-value: 3.31e-11
```

We have a high $R^2_{adj} = 0.867$ and both predictors are statistically significant, but can we already wholeheartedly trust these results? We need to perform the residual diagnostics.
Plot the estimated residuals $\hat{\epsilon}_t = \hat{Y}_t - Y_t$ vs. the observed Y_t ($t = 1, 2, \ldots, 26$ or `Year`) – see Figure 1.6. The plots show a remaining pattern, with residuals peaking, then declining. The assumption of homoskedasticity is violated. We should update the model so that this pattern is modeled or removed.

```
p1 <- ggplot(D, aes(x = Year, y = mod2$residuals)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    ylab("Residuals")
p2 <- ggplot(D, aes(x = mod2$fitted.values, y = mod2$residuals)) +
    geom_point() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    xlab("Fitted values") +
    ylab("Residuals")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

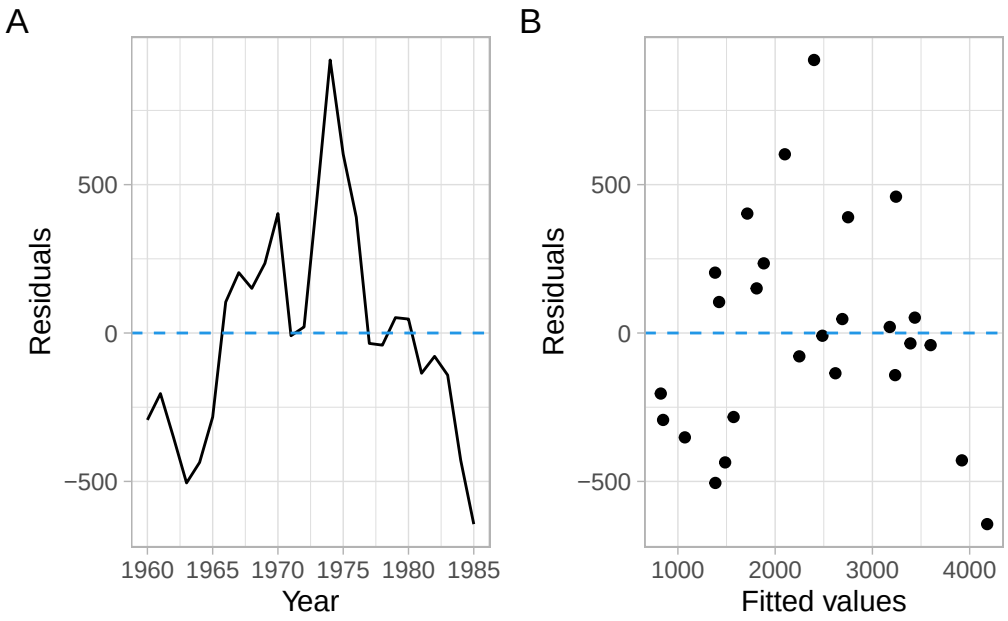


Figure 1.6.: Residuals vs. time and vs. fitted values.

Check that the residuals ϵ_t are uncorrelated (the Durbin–Watson and the runs tests):

```
lmtest::dwtest(D$DISH ~ D$RES + D$DUR)
```

```
#>
#>  Durbin-Watson test
#>
#> data:  D$DISH ~ D$RES + D$DUR
#> DW = 0.4, p-value = 7e-09
#> alternative hypothesis: true autocorrelation is greater than 0
```

```
lawstat::runs.test(mod2$residuals, plot.it = FALSE)
```

```
#>
#>  Runs Test - Two sided
#>
#> data:  mod2$residuals
#> Standardized Runs Statistic = -4, p-value = 3e-04
```

Both the tests reject H_0 of no autocorrelation; hence the assumption of uncorrelatedness is violated.
Check that the residuals ϵ_t are normally distributed using the Q-Q plot (Figure 1.7) and Shapiro–Wilk test.

```
shapiro.test(mod2$residuals)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  mod2$residuals
#> W = 1, p-value = 0.9
```

```
ggpubr::ggqqplot(mod2$residuals) +
    xlab("Standard normal quantiles")
```

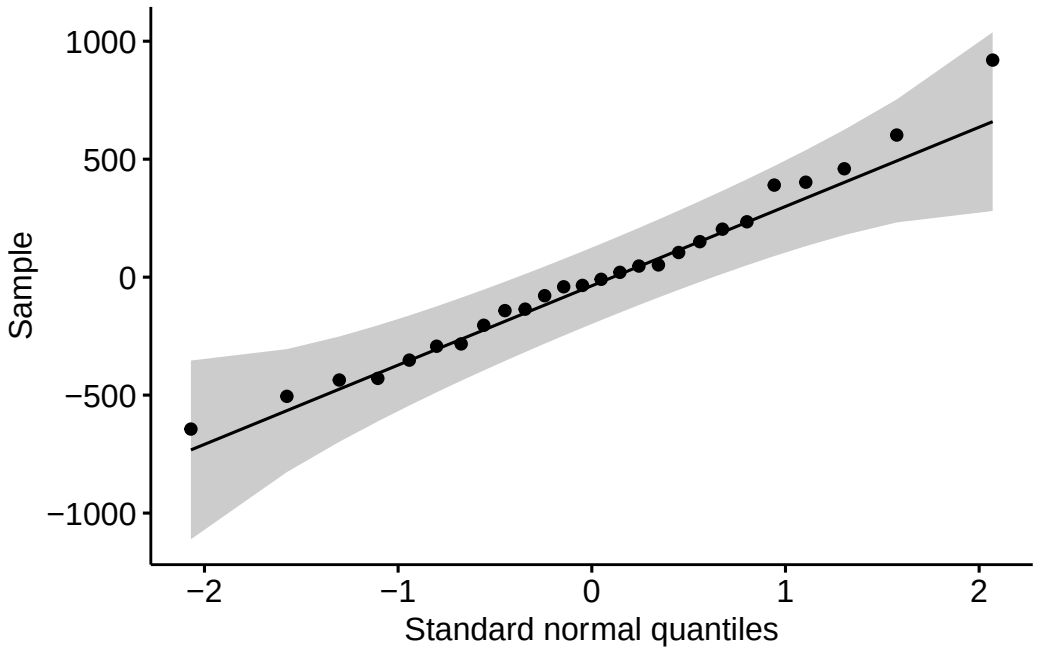


Figure 1.7.: Normal Q-Q plot of the multiple regression residuals.

Figure 1.7 and p-value of the Shapiro–Wilk test do not provide evidence against the null hypothesis of normality. The assumption of normality is satisfied.

1.2.1. Summary of the multiple linear regression residual diagnostics

1. The R^2 has been improved.
2. We do not see a visible improvement in terms of the mean and variance of the residuals.
3. The residuals are still positively correlated.
4. The residuals look normally distributed.

Even though not all OLS assumptions are satisfied we shall consider how to predict the future values of Y and to construct the prediction intervals using R.

For example, assume that we need to predict the future unit factory shipments of dishwashers (`DISH`) based on the private residential investment of 100 billion USD and durable goods expenditures of 150 billion USD.

Supply new values of independent variables and use the function `predict()`.

```
newData <- data.frame(RES = c(100), DUR = c(150))
predict(mod2, newData, se.fit = TRUE, interval = "prediction")
```

```
#> $fit
#>    fit  lwr  upr
#> 1 5559 4227 6891
#>
#> $se.fit
#> [1] 521
#>
#> $df
#> [1] 23
#>
#> $residual.scale
#> [1] 378
```

1.3. Conclusion

We have recalled the standard assumptions about residuals of linear regression models. Remember that there are some other assumptions (e.g., about linear independence of predictors) that must be verified. Refer to the reading materials for a complete list.

The methods we have used to test the homogeneity of residuals included various residual plots. The normality of residuals can be assessed using histograms or Q-Q plots and statistical tests such as the Shapiro–Wilk normality test.

The use of time series in regression presents additional ways to assess patterns in the regression residuals. A plot of residuals vs. time is assessed for homogeneity and absence of trends. Less obvious patterns, such as autocorrelation, can be tested with parametric and nonparametric tests, such as the Durbin–Watson and runs tests.

The statistical techniques we will learn aim to model or extract as much information from time series (including the autocorrelation of regression residuals) as possible, such that the remaining series are completely random.

1.4. Appendix

Alternative versions of the runs test are available, e.g., in the package `randtests`.

```
randtests::runs.test(mod2$residuals)
```

```
#>
#>  Runs Test
#>
#> data:  mod2$residuals
#> statistic = -4, runs = 5, n1 = 13, n2 = 13, n = 26, p-value = 3e-04
#> alternative hypothesis: nonrandomness
```

Difference sign test

The logic behind the difference sign test is that in a random process, there will be roughly the same number of ups (positive differences between consecutive values, i.e., $X_t - X_{t-1}$) and downs (negative differences).

Brockwell and Davis (2002): "The difference-sign test must be used with caution. A set of observations exhibiting a strong cyclic component will pass the difference-sign test for randomness since roughly half of the observations will be points of increase." We may see (Figure 1.2 and Figure 1.6), it is the case for our residuals, even though we have no cyclic component, but have a rise followed by a decline.

```
randtests::difference.sign.test(x)
```

```
#>
#>  Difference Sign Test
#>
#> data:  x
#> statistic = -2, n = 26, p-value = 0.1
#> alternative hypothesis: nonrandomness
```

```
randtests::difference.sign.test(mod1$residuals)
```

```
#>
#>  Difference Sign Test
#>
#> data:  mod1$residuals
#> statistic = 0.3, n = 26, p-value = 0.7
#> alternative hypothesis: nonrandomness
```

```
randtests::difference.sign.test(mod2$residuals)
```

```
#>
#>  Difference Sign Test
#>
#> data:  mod2$residuals
#> statistic = -0.3, n = 26, p-value = 0.7
#> alternative hypothesis: nonrandomness
```

2. Statistical Analysis & Testing

The goal of this lecture is to learn how to view time series from the frequency perspective. You should be able to recognize features of time series from a periodogram similar to how we did it using plots of the autocorrelation function (ACF).

Objectives

1. Describe the mechanics of Fourier transform for time series.
2. Present results of spectrum calculations using a periodogram or spectrogram.
3. Give examples of smoothed periodograms.
4. Demonstrate the use of the Lomb–Scargle periodogram for analysis of unequally-spaced time series.

Reading materials

- Chapter 4 in Shumway and Stoffer (2017)
- Nason (2008) on wavelet analysis

2.1. Introduction

By now, we have been working in the *time domain*, such that our analysis could be seen as a regression of the present on the past (for example, ARIMA models). We have been using many time series plots with time on the x-axis.

An alternative approach is to analyze time series in a *spectral domain*, such that use a regression of present on a linear combination of sine and cosine functions. In this type of analysis, we often use periodogram plots, with frequency or period on the x-axis.

2.2. Regression on sinusoidal components

The simplest form of spectral analysis consists of regression on a periodic component:

$$Y_t = A\cos\omega t + B\sin\omega t + C + \epsilon_t, \tag{2.1}$$

where $t = 1, \dots, T$ and $\epsilon_t \sim \text{WN}(0, \sigma^2)$. Without loss of generality, we assume $0 \leqslant \omega \leqslant \pi$. In fact, for discrete data, frequencies outside this range are aliased into this range. For example, suppose that $-\pi < (\omega = \pi - \delta) < 0$, then

$$\begin{aligned}\cos(\omega t) &= \cos((\pi - \delta)t) \\ &= \cos(\pi t)\cos(\delta t) + \sin(\pi t)\sin(\delta t) \\ &= \cos(\delta t).\end{aligned}$$

Hence, a sampled sinusoid with a frequency smaller than 0 appears to coincide with a sinusoid with frequency in the interval $[0, \pi]$.

> **Note**
>
> The term $A\cos\omega t + B\sin\omega t$ is a periodic function with the period $2\pi/\omega$. The period $2\pi/\omega$ represents the number of time units that it takes for the function to take the same value again, i.e., to complete a cycle. The frequency, measured in cycles per time unit, is given by the inverse $\omega/(2\pi)$. The angular frequency, measured in radians per time unit, is given by ω. Because of its convenience, the angular frequency ω will be used to describe the periodicity of the function, and its name is shortened to frequency when there is no danger of confusion.

Consider monthly data that exhibit a 12-month seasonality. Hence, the period $2\pi/\omega = 12$, which implies the angular frequency $\omega = \pi/6$. The frequency, measured in cycles per time unit, is given by the inverse

$$\frac{\omega}{2\pi} = \frac{1}{12} \approx 0.08.$$

The formulas to estimate parameters of regression in Equation 19.1 take a much simpler form if ω is one of the Fourier frequencies, defined by

$$\omega_j = \frac{2\pi j}{T}, \quad j = 0, \dots, \frac{T}{2},$$

then

$$\begin{aligned}
\hat{A} &= \frac{2}{T}\sum_t Y_t \cos\omega_j t, \\
\hat{B} &= \frac{2}{T}\sum_t Y_t \sin\omega_j t, \\
\hat{C} &= \overline{Y} = \frac{1}{T}\sum_t Y_t.
\end{aligned}$$

A suitable way of testing the significance of the sinusoidal component with frequency ω_j is using its contribution to the sum of squares

$$R_T(\omega_j) = \frac{T}{2}\left(\hat{A}^2 + \hat{B}^2\right).$$

If the $\epsilon_t \sim N(0, \sigma^2)$, then it follows that $\hat{A}$ and $\hat{B}$ are also independent normal, each with the variance $2\sigma^2/T$, so under the null hypothesis of $A = B = 0$ we find that

$$\frac{R_T(\omega_j)}{\sigma^2} \sim \chi^2_2$$

or equivalently that $R_T(\omega_j)/(2\sigma^2)$ has an exponential distribution with mean 1. The above theory can be extended to the simultaneous estimation of several periodic components.

2.3. Periodogram

The *Fourier transform* uses Fourier series, such as the pairs of sines and cosines with different periods, to describe the frequencies present in the original time series.

The Fourier transform applied to an equally-spaced time series Y_t (where $t = 1, \dots, T$) is also called the *discrete Fourier transform* (DFT) because the time is discrete (not the values Y_t).

To reduce the computational complexity of DFT and speed up the computations, one of the *fast Fourier transform* (FFT) algorithms is typically used.

The results of the Fourier transform are shown in a periodogram that describes the spectral properties of the signal.

The periodogram is defined as

$$I_T(\omega) = \frac{1}{2\pi T} \left| \sum_{t=1}^{T} Y_t e^{i\omega t} \right|^2 ,$$

which is an approximately unbiased estimator of the spectral density f.

Some undesirable features of the periodogram:

1. $I_T(\omega)$ for fixed ω is not a consistent estimate of $f(\omega)$, since

$$I_T(\omega_j) \sim \frac{f(\omega_j)}{2} \chi_2^2.$$

   Therefore, the variance of $f^2(\omega)$ does not tend to 0 as $T \to \infty$.
2. The independence of periodogram ordinates at different Fourier frequencies suggests that the sample periodogram plotted as a function of ω will be extremely irregular.

Suppose that $\gamma(h)$ is the autocovariance function of a stationary process and that $f(\omega)$ is the spectral density for the same process (h is the time lag and ω is the frequency). The autocovariance $\gamma(h)$ and the spectral density $f(\omega)$ are related:

$$\gamma(h) = \int_{-1/2}^{1/2} e^{2\pi i \omega h} f(\omega) d\omega,$$

and

$$f(\omega) = \sum_{h=-\infty}^{+\infty} \gamma(h) e^{-2\pi i \omega h}.$$

In the language of advanced calculus, the autocovariance and spectral density are Fourier transform pairs. These Fourier transform equations show that there is a direct link between the time domain representation and the frequency domain representation of a time series.

2.4. Smoothing

The idea behind smoothing is to take weighted averages over neighboring frequencies to reduce the variability associated with individual periodogram values. However, such an operation necessarily introduces some bias into the estimation procedure. Theoretical studies focus on the amount of smoothing that is required to obtain an optimum trade-off between bias and variance. In practice, this usually means that the choice of a kernel and amount of smoothing is somewhat subjective.

The main form of a smoothed estimator is given by

$$\hat{f}(\lambda) = \int_{-\pi}^{\pi} \frac{1}{h} K\left(\frac{\omega - \lambda}{h}\right) I_T(\omega) d\omega,$$

where $I_T(\cdot)$ is the periodogram based on T observations, $K(\cdot)$ is a kernel function, and h is the bandwidth. We usually take $K(\cdot)$ to be a nonnegative function, symmetric about 0 and integrating to 1. Thus, any symmetric density, such as the normal density, will work. In practice, however, it is more usual to take a kernel of finite range, such as the *Epanechnikov kernel*

$$K(x) = \frac{3}{4\sqrt{5}}\left(1 - \frac{x^2}{5}\right), \quad -\sqrt{5} \leqslant x \leqslant \sqrt{5},$$

which is 0 outside the interval $[-\sqrt{5}, \sqrt{5}]$. This choice of kernel function has some optimality properties. However, in practice, this optimality is less important than the choice of bandwidth h, which effectively controls the range over which the periodogram is smoothed.

There are some additional difficulties with the performance of the sample periodogram in the presence of a sinusoidal variation whose frequency is not one of the Fourier frequencies. This effect is known as *leakage*. The reason of leakage is that we always consider a truncated periodogram. Truncation implicitly assumes that the time series is periodic with a period T, which, of course, is not always true. So we artificially introduce non-existent periodicities into the estimated spectrum, i.e., cause 'leakage' of the spectrum. (If the time series is perfectly periodic over T then there is no leakage.) The leakage can be treated using an operation of tapering on the periodogram, i.e., by choosing appropriate periodogram windows.

> **Note**
>
> When we work with periodograms, we lose all phase (relative location/time origin) information: the periodogram will be the same if all the data were circularly rotated to a new time origin, i.e., the observed data are treated as perfectly periodic.

Another popular choice to smooth a periodogram is the *Daniell kernel* (with parameter m). For a time series, it is a centered moving average of values between the times $t - m$ and $t + m$ (inclusive). For example, the smoothing formula for a Daniell kernel with $m = 2$ is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2}}{5} \\ &= 0.2x_{t-2} + 0.2x_{t-1} + 0.2x_t + 0.2x_{t+1} + 0.2x_{t+2}.\end{aligned}$$

The weighting coefficients for a Daniell kernel with $m = 2$ can be checked using the function `kernel()`. In the output, the indices in `coef[ ]` refer to the time difference from the center of the average at the time t.

```
kernel("daniell", m = 2)
```

```
#> Daniell(2)
#> coef[-2] = 0.2
#> coef[-1] = 0.2
#> coef[ 0] = 0.2
#> coef[ 1] = 0.2
#> coef[ 2] = 0.2
```

The modified Daniell kernel is such that the two endpoints in the averaging receive half the weight that the interior points do. For a modified Daniell kernel with $m = 2$, the smoothing is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + 2x_{t-1} + 2x_t + 2x_{t+1} + x_{t+2}}{8} \\ &= 0.125x_{t-2} + 0.25x_{t-1} + 0.25x_t + 0.25x_{t+1} + 0.125x_{t+2}\end{aligned}$$

List the weighting coefficients:

```
kernel("modified.daniell", m = 2)
```

```
#> mDaniell(2)
#> coef[-2] = 0.125
#> coef[-1] = 0.250
#> coef[ 0] = 0.250
#> coef[ 1] = 0.250
#> coef[ 2] = 0.125
```

Either the Daniell kernel or the modified Daniell kernel can be convoluted (repeated) so that the smoothing is applied again to the smoothed values. This produces a more extensive smoothing by averaging over a wider time interval. For instance, to repeat a Daniell kernel with $m = 2$ on the smoothed values that resulted from a Daniell kernel with $m = 2$, the formula would be

$$\hat{\hat{x}}_t = \frac{\hat{x}_{t-2} + \hat{x}_{t-1} + \hat{x}_t + \hat{x}_{t+1} + \hat{x}_{t+2}}{5}.$$

The weights for averaging the original data for convoluted Daniell kernels with $m = 2$ in both smooths will be

```
kernel("daniell", m = c(2, 2))
```

```
#> Daniell(2,2)
#> coef[-4] = 0.04
#> coef[-3] = 0.08
#> coef[-2] = 0.12
#> coef[-1] = 0.16
#> coef[ 0] = 0.20
```

```
#> coef[ 1] = 0.16
#> coef[ 2] = 0.12
#> coef[ 3] = 0.08
#> coef[ 4] = 0.04
```

```
kernel("modified.daniell", m = c(2, 2))
```

```
#> mDaniell(2,2)
#> coef[-4] = 0.0156
#> coef[-3] = 0.0625
#> coef[-2] = 0.1250
#> coef[-1] = 0.1875
#> coef[ 0] = 0.2188
#> coef[ 1] = 0.1875
#> coef[ 2] = 0.1250
#> coef[ 3] = 0.0625
#> coef[ 4] = 0.0156
```

The center values are weighted slightly more heavily in the modified Daniell kernel than in the unmodified one.

When we smooth a periodogram, we are smoothing *across frequencies* rather than across times.

Example: Spectral analysis of the airline passenger data

Recall the airline passenger time series (**?@fig-airpassangers**) that has an increasing trend and strong multiplicative seasonality.
The series should be detrended before a spectral analysis. For demonstration purposes, compute periodogram for the raw data and log-transformed and detrended.

```
# Fit a quadratic trend to log-transformed data
t <- as.vector(time(AirPassengers))
mod <- lm(log10(AirPassengers) ~ poly(t, degree = 2))
res <- ts(mod$residuals)
attributes(res) <- attributes(AirPassengers)

# Compute spectra
spec_raw <- spec.pgram(AirPassengers, detrend = FALSE, plot = FALSE)
spec_log_detrend <- spec.pgram(res, detrend = FALSE, plot = FALSE)
```

The x-axes of the periodograms correspond to $\omega/(2\pi)$ or the number of cycles per the time series period specified in the `ts` object (12 months). If the frequency of 12 was not specified for this time series, the x-axes would be different, and the first spectrum peak would correspond to ≈ 0.08.
Figure 19.1 B shows that the spectrum of the raw data is dominated by low frequencies (trend). After detrending the data (Figure 19.1 C), the spectrum at frequency 1 (meaning one cycle per year) is the dominant one.

```r
pAirPassengers <- forecast::autoplot(AirPassengers, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers (thousand)") +
    ggtitle("Raw series")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
p3 <- forecast::autoplot(res, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers residuals (lg[thousand])") +
    ggtitle("Detrended and log-transformed series")
p4 <- data.frame(Spectrum = spec_log_detrend$spec,
                 Frequency = spec_log_detrend$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
(pAirPassengers + p2) / (p3 + p4) +
    plot_annotation(tag_levels = 'A')
```

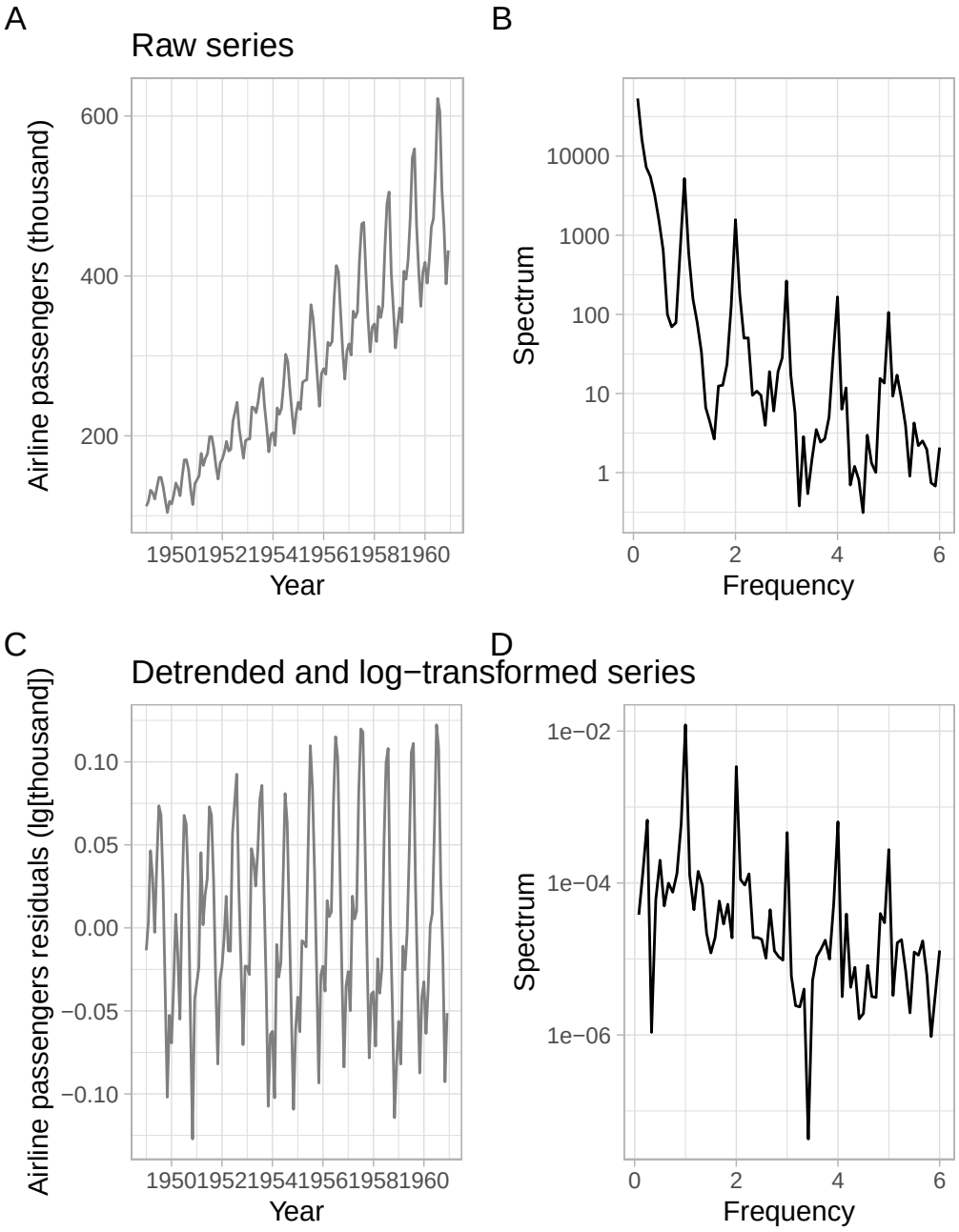


Figure 2.1.: Monthly `AirPassengers` time series, log-transformed and detrended, and the corresponding fast Fourier transforms.

Note that the function `spec.pgram()` has the option of removing a simpler (linear) trend and showing the results as a base-R plot. The confidence band in the upper right corner of this plot helps to identify the statistical significance of the peaks (Figure 19.2).

```r
par(mfrow = c(2, 2))
spec.pgram(res, spans = c(3))
spec.pgram(res, spans = c(3, 3))
spec.pgram(res, spans = c(7, 7))
spec.pgram(res, spans = c(11, 11))
```

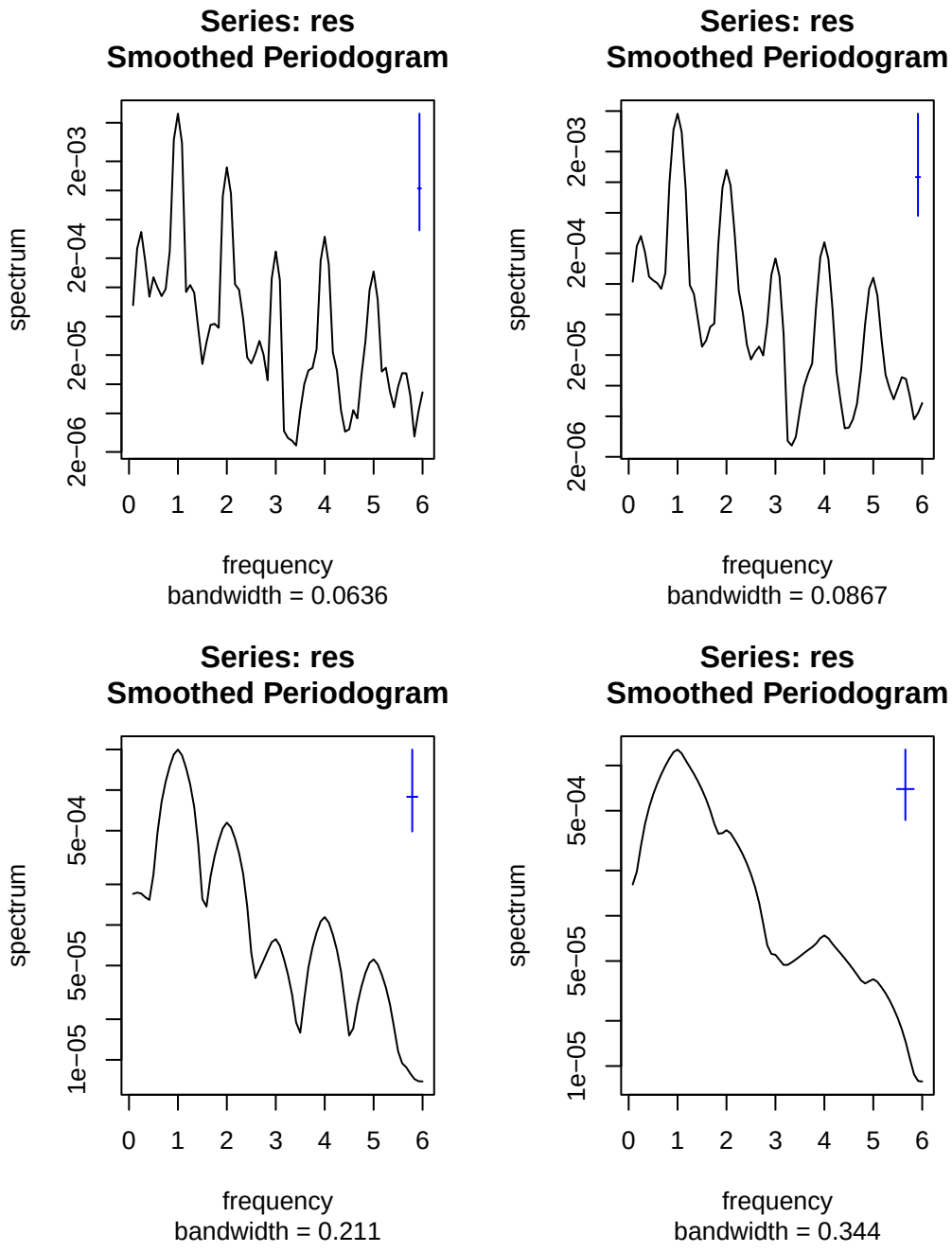


Figure 2.2.: Smoothed periodograms of monthly log-transformed and detrended `AirPassengers` time series.

Another method of testing the reality of a peak is to look at its harmonics. It is extremely unlikely that a true cycle will be shaped perfectly as a sine curve, hence at least a few of the first harmonics will show up as well. For example, in monthly data with annual seasonality (period of 12 months), we often observe peaks at 6, 4, and 3 months. This is exactly the pattern that we see in the figures above.
We can also approximate our data with an AR model and then plot the approximating periodogram of the AR model (Figure 19.3).

```
spectrum(res, method = "ar")
```

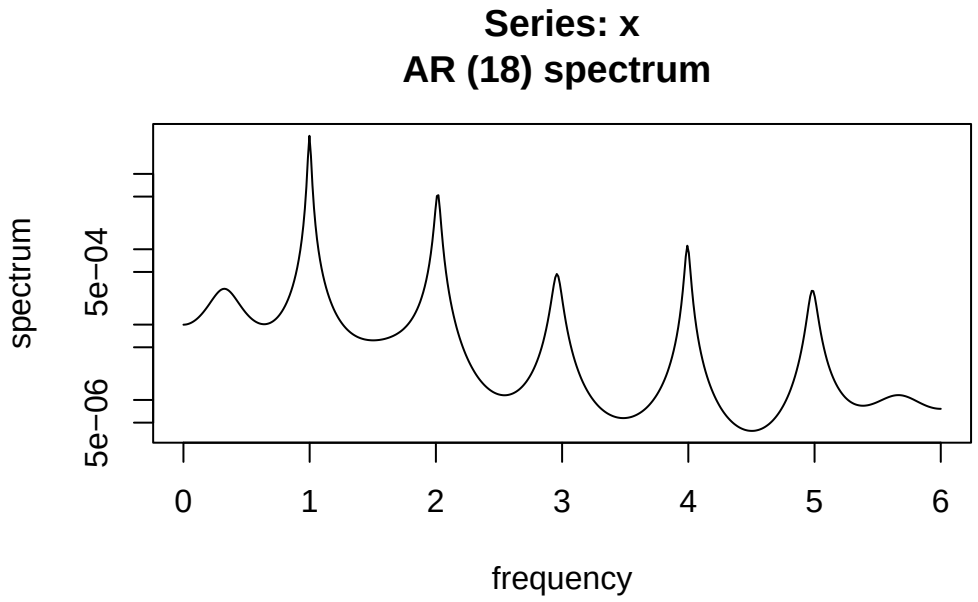


Figure 2.3.: Periodogram of AR process approximating the monthly log-transformed and detrended `AirPassengers` time series.

2.5. Periodogram for unequally-spaced time series

Unequally/unevenly-spaced time series or gaps in observations (missing data) can be handled with the Lomb–Scargle periodogram calculation. This method was originally developed for the analysis of unevenly-spaced astronomical signals (Lomb 1976; Scargle 1982) but found application in many other domains, including environmental science (e.g., Ruf 1999; Campos et al. 2008; Siskey et al. 2016).

Example: Ammonium concentration in wastewater system

For example, consider a time series `imputeTS::tsNH4` of ammonium (NH_4) concentration in a wastewater system (Figure 19.4). This time series contains observations from regular 10-minute

intervals, but some of the observations are missing.

```
forecast::autoplot(imputeTS::tsNH4) +
    xlab("Day") +
    ylab(bquote(NH[4]~concentration))
```

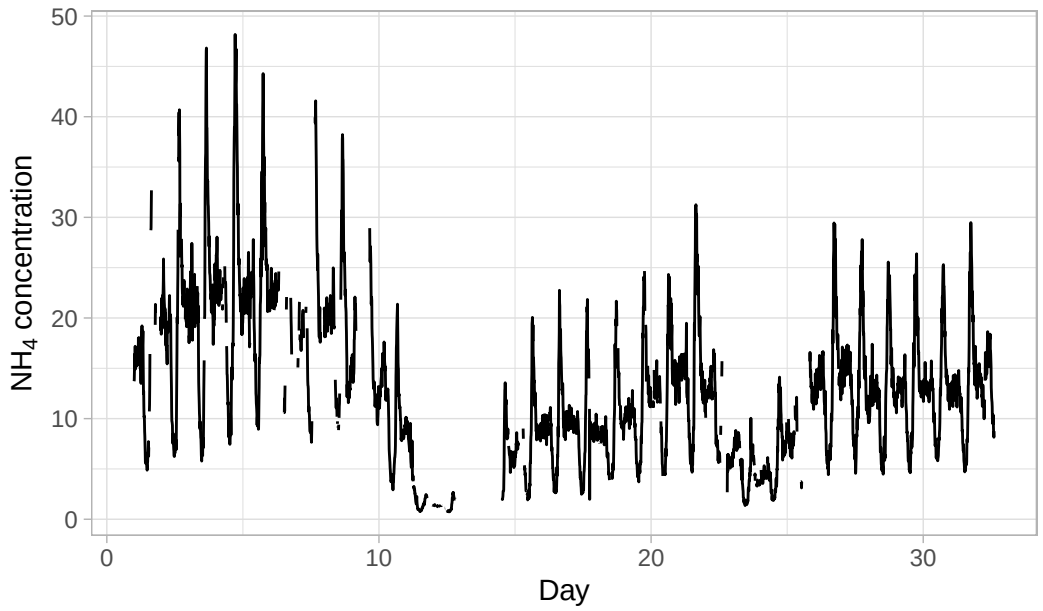


Figure 2.4.: Time series of ammonium concentration in a wastewater system.

Save the data in a format acceptable by the Lomb–Scargle function.

```
D <- data.frame(NH4 = as.vector(imputeTS::tsNH4),
                Time = as.vector(time(imputeTS::tsNH4)))
```

If needed, `na.omit(D)` can be used to remove the rows with missing values.
Figure 19.5 and Figure 19.6 show periodograms calculated based on these data. Note that the function `lomb::lsp()` has arguments adjusting the span of the frequencies and significance level.

```
par(mfrow = c(2, 2))
lomb::lsp(D$NH4, times = D$Time, type = "frequency", alpha = 0.05, main = "")
```

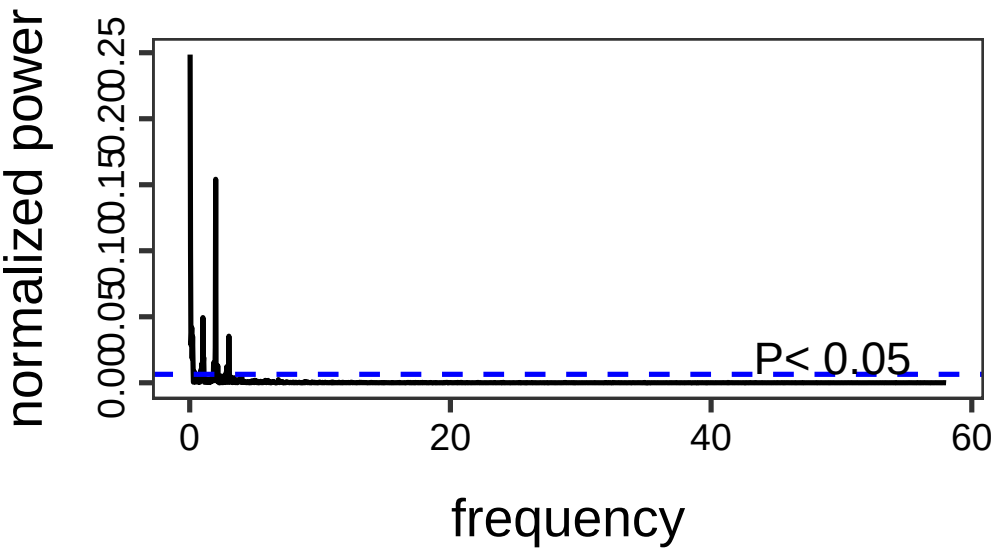


Figure 2.5.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

```
lomb::lsp(D$NH4, times = D$Time, type = "period", alpha = 0.05, main = "")
```

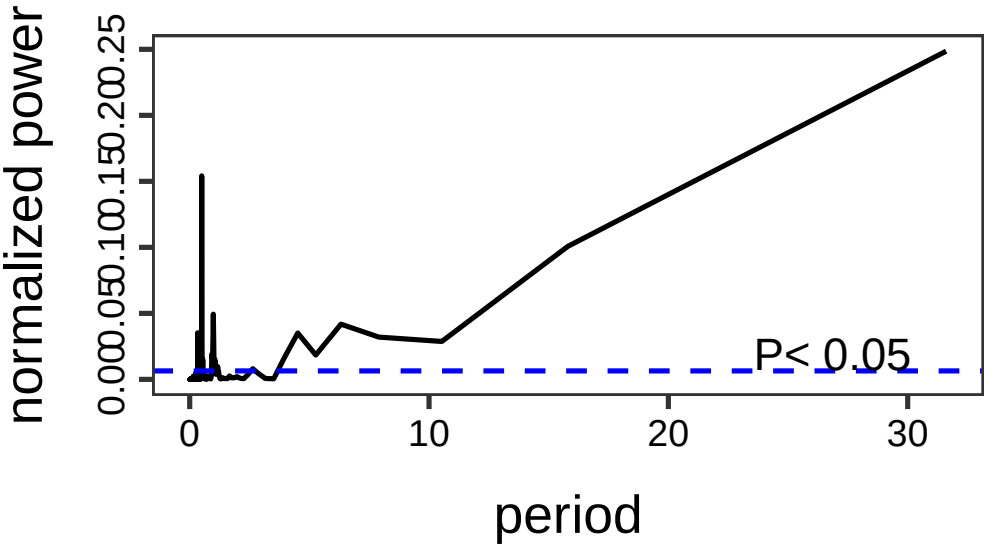


Figure 2.6.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

2.6. Frequency estimation in time

The assumption of a whole time series having a constant spectrum might be very limiting for the analysis of long time series. An estimation of frequencies locally in time allows us to study how the spectrum changes in time, and also detect smaller (short-lived) signals that would be averaged out otherwise. A straightforward solution is to separate the time series into windows (sub-periods) and estimate a spectrum in each such window.

The windowed analysis makes sense when in each window we have enough observations to estimate the frequencies. As a guide, we should have at least two observations per period to be able to represent the signal in the discrete Fourier transform. For example, we cannot estimate the seasonality if we sample once per year – this is our *sampling rate* or *sampling frequency* F_s. To start identifying seasonal periodicity, our sampling rate should be at least 2 samples per year. The highest frequency we can work with is limited by the *Nyquist frequency*

$$F_N = \frac{F_s}{2},$$

which is half of the sampling frequency.

After applying the FFT in each window, the results can be visualized in a *spectrogram*. The spectrogram combines information from multiple periodograms: it shows time on the x-axis, frequency on the y-axis, and the corresponding spectrum power as the color.

Example: Spectrograms for the NAO

Consider a long time series of the North Atlantic Oscillation (NAO) index obtained from an ensemble of 100 model-constrained NAO reconstructions (Figure 19.7).

```
NAOdf <- readr::read_csv("data/NAO.csv", skip = 12, show_col_types = FALSE) %>%
    rename(NAOproxy = nao_mc_mean) %>%
    select(Year, NAOproxy) %>%
    arrange(Year)
NAO <- ts(NAOdf$NAOproxy, start = NAOdf$Year[1])
spec_raw <- spec.pgram(NAO, detrend = FALSE, plot = FALSE, spans = 3)

# Find the frequency with the max power
fmax <- spec_raw$freq[which.max(spec_raw$spec)]
```

```r
p1 <- forecast::autoplot(NAO) +
    xlab("Year") +
    ylab("NAO")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p3 <- data.frame(Spectrum = spec_raw$spec,
                 Period = 1/spec_raw$freq) %>%
    ggplot(aes(x = Period, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = 1/fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

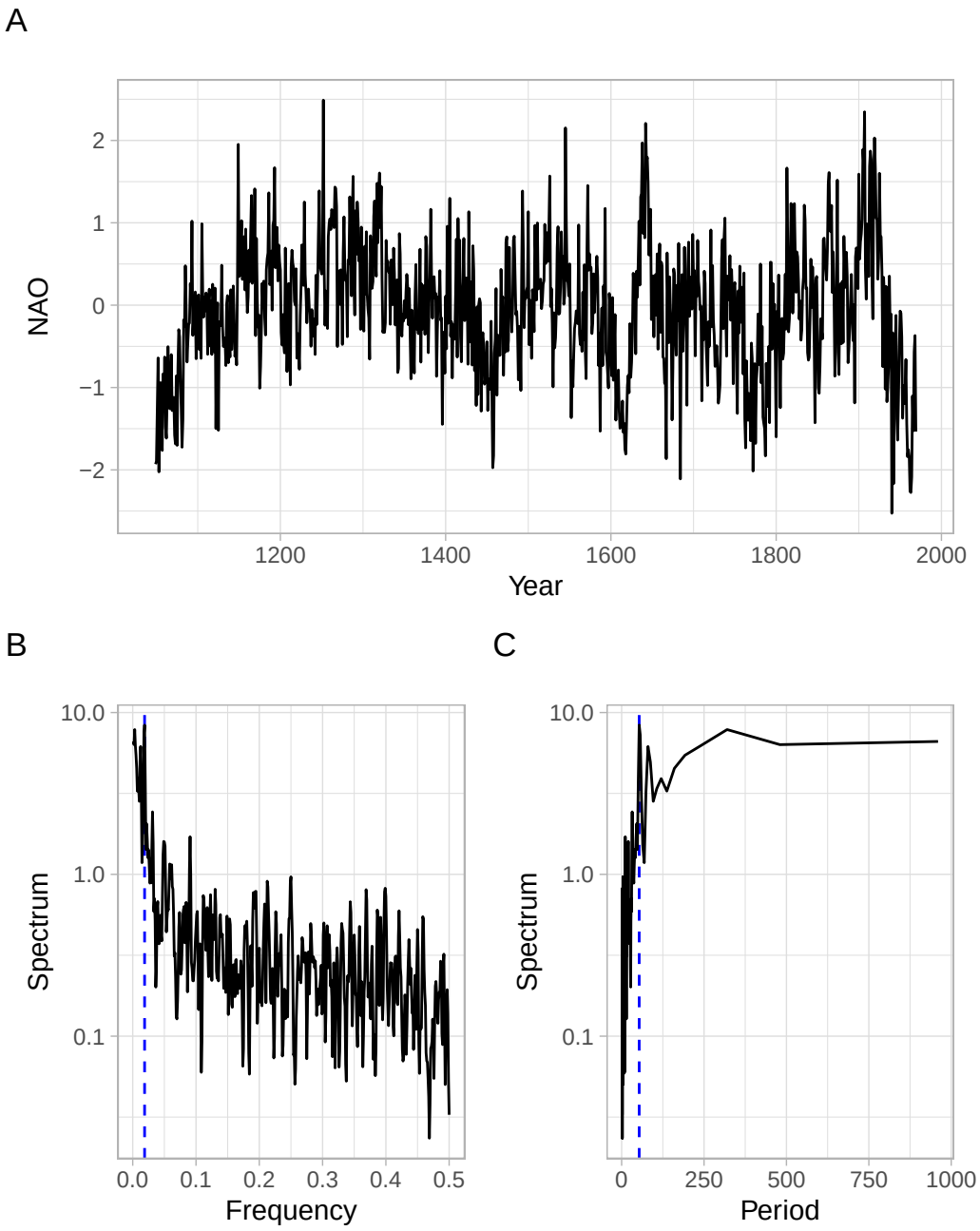


Figure 2.7.: North Atlantic Oscillation (NAO) winter index (December, January, and February) calculated as the mean of 100 model-constrained NAO reconstructions, along with its smoothed periodograms. The dashed lines denote the maximal magnitude of the spectrum.

This is an annual time series, hence the sampling frequency $F_s = 1$. From the global FFT results, the frequency with the maximal power is 0.019 year^{-1}, which is equivalent to the period of about 53.3 years. We will set the window for the FFT and calculate the spectrogram. Note

that we can use overlapping windows for a smoother picture.

```
# Set the window size (years), for applying the FFT in each window
window_size = 30

# Compute the spectrogram
SP <- signal::specgram(x = NAO,
                       n = window_size,
                       overlap = window_size/3,
                       Fs = 1)
```

See the spectrograms in Figure 19.8 and compare them with the time series plot in Figure 19.7 A.

```
par(mfrow = c(1, 2))

# Plot the spectrogram without decorations
print(SP, main = "A) Raw results")

# Discard phase information
P <- abs(SP$S)

# Normalize the spectrum to the range [0, 1]
P <- P - min(P)
P <- P/max(P)

# Extract time
Years <- time(NAO)[SP$t]

# Plot the spectrogram after processing
oce::imagep(x = Years,
            y = SP$f,
            z = t(P),
            col = oce::oce.colorsViridis,
            ylab = expression(Frequency~(y^-1 )),
            xlab = "Year",
            main = "B) After normalization and adding colors",
            drawPalette = TRUE,
            decimate = FALSE)
```

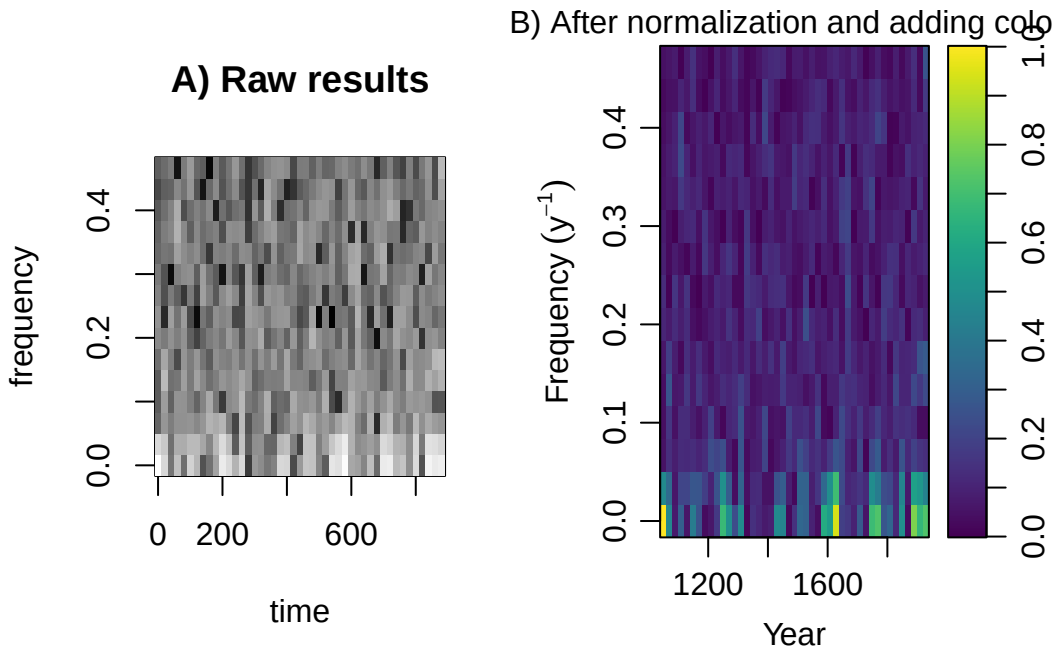


Figure 2.8.: Spectrograms for North Atlantic Oscillation (NAO) winter index.

2.7. Conclusion

For challenging problems, smoothing, multitapering, linear filtering, (repeated) pre-whitening, and Lomb–Scargle can be used together. Beware that aperiodic but autoregressive processes produce peaks in spectral densities. Harmonic analysis is a complicated art rather than a straightforward procedure.

It is extremely difficult to derive the significance of a weak periodicity from harmonic analysis. Do not believe analytical estimates (e.g., exponential probability), as they rarely apply to real data. It is essential to make simulations, typically permuting or bootstrapping the data keeping the observing times fixed. Simulations of the final model with the observation times are also advised.

2.8. Appendix

Wavelet analysis

An alternative to the windowed FFT is the wavelet analysis, which also estimates the strength of time series variations for different frequencies and times. *Wavelet* is a 'small wave' or function $\psi(t)$ approximating the variations. Popular wavelet functions are Meyer, Morlet, and Mexican hat.

Figure 19.9 shows the results of using the Morlet wavelet to process the NAO time series.

```r
library(dplR)
wv <- morlet(y1 = NAOdf$NAOproxy, x1 = NAOdf$Year)
levs <- quantile(wv$Power, probs = c(0, 0.5, 0.75, 0.9, 0.95, 1))
wavelet.plot(wv, wavelet.levels = levs)
```

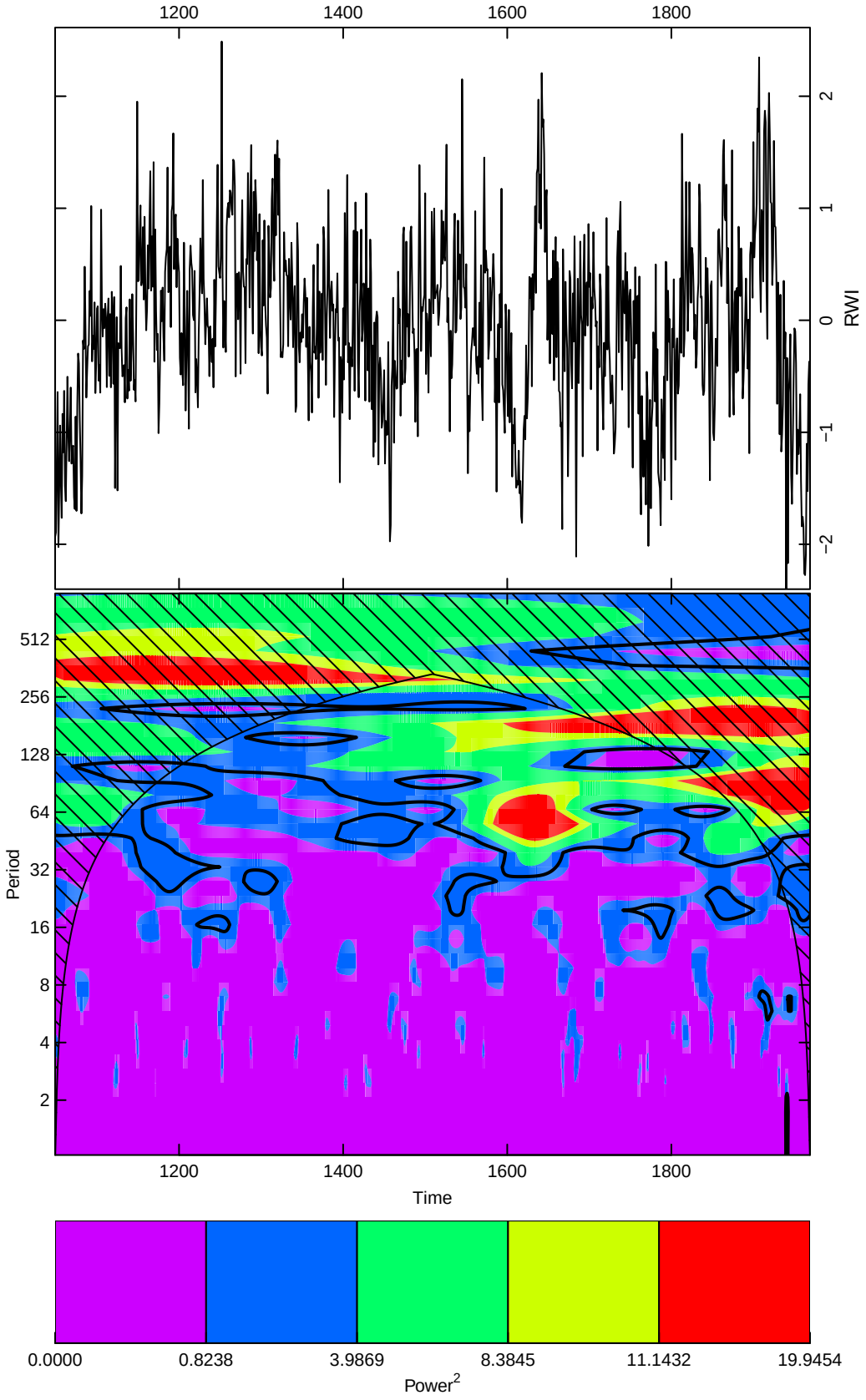


Figure 2.9.: Wavelet analysis of the North Atlantic Oscillation (NAO) winter index.

Part II.

Module 2 - Algorithms & Data Engineering

3. Data Structures with Python

4. Data Structures

Objectives

- Know which data structure to reach for given a problem's access pattern and operation requirements.
- Analyze time and space complexity for each core structure.
- Implement canonical operations from scratch: insertion, deletion, search, traversal.
- Recognize structure-to-pattern mappings in interview problems.

Reading

- *Introduction to Algorithms* — Cormen et al. (CLRS), Chapters 10–14
- *Data Structures and Algorithms in Python* — Goodrich, Tamassia, Goldwasser
- Python docs: `collections`, `heapq`, `bisect`

4.1. Big-O Complexity Reference

Complexity	Name	Example
O(1)	Constant	Hash table lookup, array index
O(log n)	Logarithmic	Binary search, balanced BST
O(n)	Linear	Array scan, linked list traversal
O(n log n)	Linearithmic	Merge sort, heap sort
$O(n^2)$	Quadratic	Nested loops, bubble sort
O(2)	Exponential	Subset generation, recursive Fibonacci
O(n!)	Factorial	Permutation enumeration

Space complexity tracks auxiliary memory. Recursive calls consume stack space — an O(n) recursive DFS uses O(n) call-stack space even with no explicit data structure.

4.2. Arrays and Lists

Python `list` is a dynamic array: a contiguous block of memory that doubles capacity on overflow.

```
arr = [1, 2, 3, 4, 5]
arr.append(6)          # O(1) amortized - occasional O(n) resize
arr.insert(2, 99)      # O(n) - shifts elements right
arr.pop()              # O(1) - remove last
arr.pop(2)             # O(n) - remove at index, shift left
arr[2]                 # O(1) - random access
```

4.2.1. Prefix Sums

Precompute cumulative sums to answer range-sum queries in O(1) after O(n) preprocessing.

```
def build_prefix(arr):
    prefix = [0] * (len(arr) + 1)
    for i, v in enumerate(arr):
        prefix[i + 1] = prefix[i] + v
    return prefix

def range_sum(prefix, l, r):  # inclusive [l, r]
    return prefix[r + 1] - prefix[l]

arr = [1, 3, 5, 7, 9]
prefix = build_prefix(arr)
print(range_sum(prefix, 1, 3))  # 3+5+7 = 15
```

4.2.2. Complexity

Operation	Time	Notes
Access by index	O(1)	
Append	O(1) amortized	O(n) worst on resize
Insert / Delete at index	O(n)	Shifts elements
Search (unsorted)	O(n)	
Search (sorted)	O(log n)	Binary search

4.3. Hash Tables

A hash table maps keys to values via a hash function. Python `dict` uses open addressing with a load factor threshold.

Collision resolution: Python uses probing; Java's `HashMap` uses chaining. **Worst case:** O(n) if all keys collide — practically O(1) with a good hash function.

```
from collections import defaultdict, Counter, OrderedDict

# Frequency counting
freq = Counter("abracadabra")
freq.most_common(2)         # [('a', 5), ('b', 2)]

# Grouping
groups = defaultdict(list)
for word in ["cat", "act", "tac", "dog", "god"]:
    key = tuple(sorted(word))
    groups[key].append(word)
# {('a','c','t'): ['cat','act','tac'], ('d','g','o'): ['dog','god']}

# Two-sum pattern
def two_sum(nums, target):
    seen = {}
    for i, n in enumerate(nums):
        complement = target - n
        if complement in seen:
            return [seen[complement], i]
        seen[n] = i
```

4.3.1. When to Use Sets vs Dicts

```
# Set: O(1) membership, no values needed
valid = {"admin", "editor", "viewer"}
role = "admin"
if role in valid:        # O(1)
    pass

# Dict: O(1) lookup when value matters
cache = {}
cache["key"] = compute_value()
result = cache.get("key", default_value)
```

4.3.2. Complexity

Operation	Average	Worst
Insert	O(1)	O(n)
Lookup	O(1)	O(n)
Delete	O(1)	O(n)
Iteration	O(n)	O(n)

4.4. Stacks and Queues

4.4.1. Stack (LIFO)

Use `list` or `collections.deque`. Stack is natural for: bracket matching, undo/redo, DFS iteratively, expression evaluation.

```
stack = []
stack.append(1); stack.append(2); stack.append(3)
stack.pop()   # 3

# Valid parentheses
def is_valid(s):
    stack = []
    pairs = {')': '(', '}': '{', ']': '['}
    for ch in s:
        if ch in "([{":
            stack.append(ch)
        elif not stack or stack[-1] != pairs[ch]:
            return False
        else:
            stack.pop()
    return not stack
```

4.4.2. Monotonic Stack

Maintains a stack in sorted (monotone) order. Solves "next greater / smaller element" in O(n).

```
def next_greater(arr):
    result = [-1] * len(arr)
    stack = []  # stores indices
    for i, v in enumerate(arr):
        while stack and arr[stack[-1]] < v:
            result[stack.pop()] = v
        stack.append(i)
    return result
```

```
next_greater([2, 1, 2, 4, 3])  # [4, 2, 4, -1, -1]
```

4.4.3. Queue (FIFO)

Use `collections.deque` — O(1) append and popleft. Natural for BFS, sliding window, task scheduling.

```
from collections import deque

queue = deque()
queue.append(1); queue.append(2)
queue.popleft()   # 1 - O(1)

# Sliding window maximum (monotonic deque)
def max_sliding_window(nums, k):
    dq, result = deque(), []
    for i, n in enumerate(nums):
        while dq and nums[dq[-1]] < n:
            dq.pop()
        dq.append(i)
        if dq[0] == i - k:    # out of window
            dq.popleft()
        if i >= k - 1:
            result.append(nums[dq[0]])
    return result
```

4.4.4. Priority Queue (Heap)

Python's `heapq` is a **min-heap**. For max-heap, negate values.

```
import heapq

# Min-heap
heap = [5, 3, 1, 4, 2]
heapq.heapify(heap)          # O(n)
heapq.heappush(heap, 0)      # O(log n)
heapq.heappop(heap)          # 0 - O(log n)

# K largest elements
def k_largest(nums, k):
    return heapq.nlargest(k, nums)   # O(n log k)

# K-way merge
def merge_k_sorted(lists):
    heap = [(lst[0], i, 0) for i, lst in enumerate(lists) if lst]
```

```
    heapq.heapify(heap)
    result = []
    while heap:
        val, i, j = heapq.heappop(heap)
        result.append(val)
        if j + 1 < len(lists[i]):
            heapq.heappush(heap, (lists[i][j + 1], i, j + 1))
    return result
```

4.5. Linked Lists

Each node stores data and a pointer. No random access — O(n) search — but O(1) insert/delete at a known position.

```
class ListNode:
    def __init__(self, val=0, nxt=None):
        self.val = val
        self.next = nxt

# Reverse a linked list - iterative
def reverse(head):
    prev, curr = None, head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev, curr = curr, nxt
    return prev

# Detect cycle - Floyd's two-pointer
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True
    return False

# Find middle - slow/fast pointers
def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
```

```
        fast = fast.next.next
    return slow
```

4.5.1. Dummy Node Pattern

Dummy/sentinel nodes simplify edge cases (empty list, head deletion).

```
def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    fast = slow = dummy
    for _ in range(n + 1):
        fast = fast.next
    while fast:
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next
    return dummy.next
```

4.6. Trees

4.6.1. Binary Tree Traversals

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Recursive traversals
def inorder(root):    # left → root → right  (sorted for BST)
    return inorder(root.left) + [root.val] + inorder(root.right) if root else []

def preorder(root):   # root → left → right
    return [root.val] + preorder(root.left) + preorder(root.right) if root else []

def postorder(root):  # left → right → root
    return postorder(root.left) + postorder(root.right) + [root.val] if root else []

# Level-order (BFS)
from collections import deque
def level_order(root):
```

```
    if not root: return []
    q, result = deque([root]), []
    while q:
        level = []
        for _ in range(len(q)):
            node = q.popleft()
            level.append(node.val)
            if node.left:  q.append(node.left)
            if node.right: q.append(node.right)
        result.append(level)
    return result
```

4.6.2. Binary Search Tree (BST)

BST invariant: `left.val < node.val < right.val`. Search, insert, delete: O(log n) average, O(n) worst (degenerate/skewed tree).

```
def search(root, target):
    if not root or root.val == target:
        return root
    return search(root.left, target) if target < root.val else search(root.right, target)

def insert(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insert(root.left, val)
    else:
        root.right = insert(root.right, val)
    return root

def height(root):
    return 0 if not root else 1 + max(height(root.left), height(root.right))

def is_valid_bst(root, lo=float('-inf'), hi=float('inf')):
    if not root: return True
    if not (lo < root.val < hi): return False
    return is_valid_bst(root.left, lo, root.val) and is_valid_bst(root.right, root.val, hi)
```

4.6.3. Heap (Complete Binary Tree)

A heap is a complete binary tree stored as an array. Min-heap: `parent  children`.

```
# Array representation: parent(i) = (i-1)//2, left(i) = 2i+1, right(i) = 2i+2

class MinHeap:
    def __init__(self): self.h = []

    def push(self, val):
        self.h.append(val)
        self._sift_up(len(self.h) - 1)

    def pop(self):
        self.h[0], self.h[-1] = self.h[-1], self.h[0]
        val = self.h.pop()
        if self.h: self._sift_down(0)
        return val

    def _sift_up(self, i):
        while i > 0:
            p = (i - 1) // 2
            if self.h[p] > self.h[i]:
                self.h[p], self.h[i] = self.h[i], self.h[p]
                i = p
            else: break

    def _sift_down(self, i):
        n = len(self.h)
        while True:
            smallest = i
            for c in [2*i+1, 2*i+2]:
                if c < n and self.h[c] < self.h[smallest]:
                    smallest = c
            if smallest == i: break
            self.h[i], self.h[smallest] = self.h[smallest], self.h[i]
            i = smallest
```

4.7. Tries (Prefix Trees)

Efficient for prefix search, autocomplete, and dictionary lookups. O(m) operations where m = word length.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False
```

```python
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:
            node = node.children.setdefault(ch, TrieNode())
        node.is_end = True

    def search(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children: return False
            node = node.children[ch]
        return node.is_end

    def starts_with(self, prefix):
        node = self.root
        for ch in prefix:
            if ch not in node.children: return False
            node = node.children[ch]
        return True
```

Interview applications: autocomplete, word search in a grid (with DFS), longest common prefix, word break problem.

4.8. Graphs

A graph G = (V, E). Two standard representations:

```python
# Adjacency list - sparse graphs (most interview problems)
graph = {
    0: [1, 2],
    1: [0, 3],
    2: [0, 4],
    3: [1],
    4: [2]
}

# Adjacency matrix - dense graphs or when edge weight lookup needed
n = 5
```

```
matrix = [[0]*n for _ in range(n)]
matrix[0][1] = 1   # directed edge 0 → 1
```

4.8.1. BFS — Shortest path in unweighted graph

```
from collections import deque

def bfs(graph, start):
    visited = {start}
    queue = deque([(start, 0)])    # (node, distance)
    while queue:
        node, dist = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, dist + 1))
```

4.8.2. DFS — Connected components, cycle detection, topological sort

```
def dfs(graph, node, visited=None):
    if visited is None: visited = set()
    visited.add(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Count connected components
def count_components(n, edges):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    visited = set()
    count = 0
    for node in range(n):
        if node not in visited:
            dfs(graph, node, visited)
            count += 1
    return count
```

4.8.3. Topological Sort (Kahn's Algorithm — BFS)

```
from collections import deque

def topological_sort(n, prerequisites):
    graph = defaultdict(list)
    indegree = [0] * n
    for u, v in prerequisites:
        graph[v].append(u)
        indegree[u] += 1
    queue = deque([i for i in range(n) if indegree[i] == 0])
    order = []
    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)
    return order if len(order) == n else []    # empty = cycle detected
```

4.9. Union-Find (Disjoint Set Union)

Efficiently tracks connected components. With path compression + union by rank: O((n)) O(1) per operation.

Use: detect cycles in undirected graphs, Kruskal's MST, network connectivity, grouping problems.

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.components = n

    def find(self, x):                          # path compression
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):                      # union by rank
        px, py = self.find(x), self.find(y)
        if px == py: return False               # already connected
        if self.rank[px] < self.rank[py]:
```

```
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        self.components -= 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

4.10. Core Patterns

4.10.1. Two Pointers

```
# Pair sum in sorted array
def pair_sum(arr, target):
    l, r = 0, len(arr) - 1
    while l < r:
        s = arr[l] + arr[r]
        if s == target:   return (arr[l], arr[r])
        elif s < target:  l += 1
        else:             r -= 1
    return None

# Remove duplicates in-place
def remove_duplicates(nums):
    k = 1
    for i in range(1, len(nums)):
        if nums[i] != nums[k - 1]:
            nums[k] = nums[i]
            k += 1
    return k
```

4.10.2. Sliding Window

Fixed or variable window over a sequence. Avoids $O(n^2)$ nested loops.

```
# Fixed window - max sum of k elements
def max_sum_k(arr, k):
    window = sum(arr[:k])
    best = window
    for i in range(k, len(arr)):
```

```
        window += arr[i] - arr[i - k]
        best = max(best, window)
    return best

# Variable window - longest substring with at most k distinct chars
def longest_k_distinct(s, k):
    freq = defaultdict(int)
    l = best = 0
    for r, ch in enumerate(s):
        freq[ch] += 1
        while len(freq) > k:
            freq[s[l]] -= 1
            if freq[s[l]] == 0: del freq[s[l]]
            l += 1
        best = max(best, r - l + 1)
    return best
```

4.10.3. Fast & Slow Pointers

```
# Find cycle start in linked list
def cycle_start(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            slow = head
            while slow is not fast:
                slow = slow.next
                fast = fast.next
            return slow
    return None
```

4.11. Structure Selection Guide

Problem Signal	Best Structure
Fast lookup by key	`dict` / hash set
Top-k / priority	`heapq`
LIFO / undo / matching parens	Stack (`list`)
FIFO / BFS	`deque`
Prefix search / autocomplete	Trie

Problem Signal	Best Structure
Sorted order + dynamic inserts	Sorted list (`sortedcontainers`) / BST
Connectivity / components	Union-Find
Shortest path (unweighted)	BFS
Shortest path (weighted)	Dijkstra (`heapq`)
Dependencies / ordering	Topological sort
Range sum queries	Prefix sums / Fenwick tree

Interview Focus

Derive the amortized O(1) cost of dynamic array append. Implement a min-heap from scratch. Detect a cycle in a linked list without extra space. Implement Trie insert and search. Explain Union-Find with path compression and union by rank — what is the amortized complexity?

5. Algorithms & Coding Patterns

Objectives

After this chapter, you should be able to recognize which algorithmic pattern applies to a problem before writing a single line of code, derive time and space complexity from first principles, and explain the intuition behind every pattern clearly enough to teach it.

Reading

- *Introduction to Algorithms* — Cormen, Leiserson, Rivest, Stein (CLRS), Chapters 2–4, 15–16, 22–25
- *Algorithm Design* — Kleinberg & Tardos
- *Grokking the Coding Interview* — Design Gurus

5.1. Introduction to Coding Patterns

The most important skill in algorithmic problem solving is not memorizing solutions — it is recognizing which *pattern* a problem belongs to before you begin.

Every problem has a signal. A sorted array signals binary search. A contiguous window signals sliding window. Overlapping subproblems signal dynamic programming. Experienced engineers recognize these signals within the first thirty seconds of reading a problem; this chapter is designed to build that recognition from the ground up.

The patterns in this chapter are organized by the structural insight that makes each one work:

Pattern	Core Insight	Complexity Gain
Two Pointer	Eliminate half the search space per step	$O(n^2) \rightarrow O(n)$
Prefix Sum	Answer range queries with precomputed sums	$O(n)$ per query $\rightarrow O(1)$
Tortoise & Hare	Detect cycles without extra space	$O(n)$ time, $O(1)$ space
Sliding Window	Extend/shrink a window instead of restarting	$O(n^2) \rightarrow O(n)$
Two Pass	One pass builds state; a second pass uses it	Enables $O(n)$ solutions
Bit Manipulation	Exploit binary representation for $O(1)$ tricks	Constant factor speedups

Pattern	Core Insight	Complexity Gain
Cyclic Sort	Use index as identity for in-place placement	O(n) for bounded integers

Each pattern exists because brute force is too slow. The pattern is the minimal structural insight that eliminates the waste.

5.2. Two Pointer

5.2.1. Intuition

Given a sorted array, a naive search for a pair summing to a target takes $O(n^2)$ — try every pair. The two-pointer insight is that sorting imposes *order*, and order lets us eliminate candidates without examining them.

Place one pointer at the left (smallest element) and one at the right (largest). Compute their sum:

- **Sum $<$ target**: the left element is too small. Any pair containing it with the current right or anything to its left will also be too small. Move the left pointer right.
- **Sum $>$ target**: the right element is too large. Move the right pointer left.
- **Sum $=$ target**: found.

Each step eliminates at least one element from consideration. The two pointers together traverse at most n steps. **$O(n^2)$ becomes $O(n)$**.

5.2.2. Template

```
def two_pointer(arr):
    arr.sort()              # O(n log n) - prerequisite
    left, right = 0, len(arr) - 1
    while left < right:
        val = f(arr[left], arr[right])
        if val == target:
            # record result
            left += 1; right -= 1
        elif val < target:
            left += 1       # need larger value
        else:
            right -= 1      # need smaller value
```

5.2.3. Worked Example: Three Sum

Find all unique triplets in `nums` that sum to zero. Brute force is $O(n^3)$. Two pointers reduce it to $O(n^2)$.

Fix one element `nums[i]` with a loop. The problem then reduces to *two sum* on the remaining subarray — exactly what two pointers solve.

```
def three_sum(nums):
    nums.sort()
    result = []
    for i in range(len(nums) - 2):
        if i > 0 and nums[i] == nums[i - 1]:
            continue                            # skip duplicates at the fixed element
        left, right = i + 1, len(nums) - 1
        while left < right:
            s = nums[i] + nums[left] + nums[right]
            if s == 0:
                result.append([nums[i], nums[left], nums[right]])
                while left < right and nums[left]  == nums[left + 1]: left += 1
                while left < right and nums[right] == nums[right - 1]: right -= 1
                left += 1; right -= 1
            elif s < 0:
                left += 1
            else:
                right -= 1
    return result
```

Complexity: $O(n \log n)$ for sort + $O(n^2)$ for the outer loop × inner two-pointer scan = $\mathbf{O(n^2)}$ overall.

5.2.4. When to Apply

Recognize two pointers when the input is sorted (or can be sorted) and the problem asks for a *pair*, *triplet*, or *partition* satisfying a condition on values. Also applies to in-place problems like removing duplicates or squaring a sorted array.

5.3. Prefix Sum

5.3.1. Intuition

A *range query* asks: what is the sum (or product, XOR, count) of elements from index `l` to `r`? A naive loop answers each query in $O(n)$. If there are Q queries, that is $O(nQ)$.

The prefix sum insight is that `sum(l, r) = prefix[r+1] - prefix[l]`. By precomputing prefix sums in O(n), every query becomes a single subtraction — **O(1) per query**.

Formally, define:

$$P[0] = 0, \quad P[i] = \sum_{k=0}^{i-1} a[k]$$

Then $\text{sum}(l, r) = P[r+1] - P[l]$.

5.3.2. Construction and Query

```
def build_prefix(arr):
    n = len(arr)
    prefix = [0] * (n + 1)
    for i in range(n):
        prefix[i + 1] = prefix[i] + arr[i]
    return prefix

def range_sum(prefix, l, r):    # inclusive [l, r]
    return prefix[r + 1] - prefix[l]
```

5.3.3. 2D Prefix Sum

The same idea extends to matrices. Define `P[i][j]` as the sum of all elements in the rectangle from (0,0) to (i-1, j-1). An arbitrary rectangle query then requires only four lookups:

$$\text{sum}(r_1, c_1, r_2, c_2) = P[r_2+1][c_2+1] - P[r_1][c_2+1] - P[r_2+1][c_1] + P[r_1][c_1]$$

This is inclusion-exclusion: add the full rectangle, subtract the two over-excluded strips, add back the doubly-excluded corner.

```
def build_prefix_2d(matrix):
    m, n = len(matrix), len(matrix[0])
    P = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            P[i][j] = (matrix[i-1][j-1]
                       + P[i-1][j] + P[i][j-1] - P[i-1][j-1])
    return P
```

5.3.4. When to Apply

Reach for prefix sums whenever you see repeated range queries over a static array. Also look for problems involving "subarray sum equals k" — pairing a prefix sum with a hash map gives an O(n) solution.

```
def subarray_sum_k(nums, k):
    # Count subarrays whose sum equals k
    count = prefix = 0
    seen = {0: 1}       # prefix sum → frequency
    for n in nums:
        prefix += n
        count += seen.get(prefix - k, 0)
        seen[prefix] = seen.get(prefix, 0) + 1
    return count
```

5.4. Tortoise & Hare

5.4.1. Intuition

Floyd's cycle detection algorithm answers: *does a sequence contain a cycle, and if so, where does it begin?* The naive approach uses a visited set — O(n) space. Floyd's algorithm uses two pointers moving at different speeds — **O(n) time, O(1) space**.

The key observation is that in a cyclic sequence, a *fast* pointer (moving two steps at a time) and a *slow* pointer (moving one step) will inevitably meet inside the cycle. When they meet, the distance from the head to the cycle start equals the distance from the meeting point to the cycle start — a geometric fact that can be derived algebraically.

Derivation. Let F = distance from head to cycle start, C = cycle length, and k = distance from cycle start to meeting point. When slow has traveled $F + k$ steps, fast has traveled $2(F + k)$ steps. Fast has also traveled an integer number of full cycles beyond where slow is:

$$2(F + k) - (F + k) = F + k = nC$$

for some integer n. Therefore $F = nC - k$: the distance from the head to the cycle start equals $nC - k$, which is the same as the distance from the meeting point back to the cycle start (going forward in the cycle). Moving one pointer to the head and walking both forward at the same speed will find the cycle start.

5.4.2. Implementation

```python
def find_cycle_start(head):
    slow = fast = head

    # Phase 1: detect a cycle
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            break
    else:
        return None                 # no cycle

    # Phase 2: find cycle start
    slow = head
    while slow is not fast:
        slow = slow.next
        fast = fast.next
    return slow                     # cycle start node
```

5.4.3. When to Apply

Any problem involving an implicit sequence (not necessarily a linked list) that might revisit a state. Detecting duplicate numbers in an array where values are bounded by the array length is structurally a linked-list cycle problem in disguise.

5.5. Sliding Window

5.5.1. Intuition

Many array and string problems ask for an *optimal contiguous subarray* — longest, shortest, maximum sum, or minimum containing a target set. A brute force approach considers every $O(n^2)$ pair (i, j) and recomputes the window sum in $O(n)$, giving $O(n^3)$ total. Prefix sums reduce this to $O(n^2)$.

The sliding window insight is stronger: maintain the window state *incrementally*. When the right boundary advances by one, update the state by adding one element. When the left boundary advances, remove one element. No recomputation — each element is added and removed at most once, giving $\mathbf{O(n)}$ total.

Two variants exist:

- **Fixed window:** size k is given. Advance both pointers together.
- **Variable window:** expand right until a condition is met, then shrink from the left.

5.5.2. Fixed Window

```python
def max_sum_subarray_k(arr, k):
    """Maximum sum of any contiguous subarray of length k."""
    window_sum = sum(arr[:k])
    best = window_sum
    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i - k]   # add new, remove old
        best = max(best, window_sum)
    return best
```

5.5.3. Variable Window

The variable window expands greedily and contracts only when a constraint is violated. The contraction uses a *while* loop, not an *if* — keep contracting until the constraint is restored.

```python
def longest_substring_k_distinct(s, k):
    """Longest substring with at most k distinct characters."""
    from collections import defaultdict
    freq = defaultdict(int)
    left = best = 0
    for right, ch in enumerate(s):
        freq[ch] += 1
        while len(freq) > k:                # constraint violated
            freq[s[left]] -= 1
            if freq[s[left]] == 0:
                del freq[s[left]]
            left += 1
        best = max(best, right - left + 1)
    return best
```

5.5.4. Minimum Window Substring

A classic hard problem: find the shortest substring of `s` containing all characters of `t`. The key is tracking how many characters still *need* to be matched.

```python
from collections import Counter

def min_window(s, t):
    need = Counter(t)
    missing = len(t)            # characters still needed
    best = ""
    left = 0
    for right, ch in enumerate(s):
        if need[ch] > 0:
```

```
            missing -= 1
        need[ch] -= 1
        if missing == 0:        # window contains all of t
            while need[s[left]] < 0:    # shrink from left
                need[s[left]] += 1
                left += 1
            if not best or right - left + 1 < len(best):
                best = s[left:right + 1]
            need[s[left]] += 1          # release leftmost
            missing += 1
            left += 1
    return best
```

Complexity: O(|s| + |t|). Each character of `s` is processed at most twice — once when `right` passes it, once when `left` passes it.

5.5.5. When to Apply

The word "contiguous subarray" or "substring" combined with an optimization (longest, shortest, maximum, minimum) almost always signals a sliding window. The fixed variant applies when the window size is given; the variable variant applies when the window size itself is what you are optimizing.

5.6. Two Pass

5.6.1. Intuition

Some problems require information from *both sides* of each element — for example, the product of all elements to the left and all elements to the right. A single forward pass cannot capture the right-side context. The two-pass pattern solves this by making one pass in each direction, accumulating running state.

The two passes need not be in opposite directions; sometimes both passes are forward, with the second pass using state built by the first.

5.6.2. Worked Example: Product of Array Except Self

Without division, compute an output array where `output[i]` is the product of all elements of `nums` except `nums[i]`.

- **Pass 1 (left to right):** for each position, store the product of all elements to its left.
- **Pass 2 (right to left):** multiply by the running product of all elements to its right.

```python
def product_except_self(nums):
    n = len(nums)
    output = [1] * n

    # Pass 1: left products
    left_product = 1
    for i in range(n):
        output[i] = left_product
        left_product *= nums[i]

    # Pass 2: multiply by right products
    right_product = 1
    for i in range(n - 1, -1, -1):
        output[i] *= right_product
        right_product *= nums[i]

    return output
```

Complexity: O(n) time, O(1) auxiliary space (the output array is not counted as extra space by convention).

5.6.3. When to Apply

Look for problems where each element's answer depends on context from *both directions* — trapping rainwater, candy distribution, and histogram-area problems all have this structure. The two-pass pattern cleanly separates left-context accumulation from right-context accumulation.

5.7. Bit Manipulation

5.7.1. Intuition

Every integer is a sequence of bits. Operations on bits are O(1) and extremely fast at the hardware level. Bit manipulation exploits this to replace loops, auxiliary data structures, or arithmetic with constant-time bitwise operations.

5.7.2. Essential Operations

Expression	Meaning	Use
`x & (x - 1)`	Clear the lowest set bit	Count set bits, power-of-two check

Expression	Meaning	Use
`x & (-x)`	Isolate the lowest set bit	Fenwick tree, LSB extraction
`x ^ x == 0`	XOR with itself cancels	Find the unique element
`1 << k`	The value 2^k	Bit masks
`x >> k`	Arithmetic right shift by k	Divide by 2^k

5.7.3. Core Identities

XOR cancellation: $a \oplus a = 0$ and $a \oplus 0 = a$. This means XOR-ing a multiset where every element appears twice except one leaves only the unique element.

```
import functools

def single_number(nums):
    return functools.reduce(lambda a, b: a ^ b, nums)
```

Counting set bits — Brian Kernighan's algorithm: The expression `x & (x - 1)` clears the lowest set bit of x. Repeating this until x becomes zero counts the number of set bits in exactly as many iterations as there are set bits — never more.

```
def popcount(n):
    count = 0
    while n:
        n &= n - 1      # remove lowest set bit
        count += 1
    return count
```

Power of two check: A power of two has exactly one set bit, so `x & (x - 1) == 0` (and `x > 0`).

Generating all subsets: The integers from 0 to $2^n - 1$ enumerate all n-bit masks. Each mask corresponds to a subset.

```
def all_subsets(nums):
    n = len(nums)
    subsets = []
    for mask in range(1 << n):          # 0 to 2^n - 1
        subset = [nums[i] for i in range(n) if mask & (1 << i)]
        subsets.append(subset)
    return subsets
```

5.7.4. When to Apply

Bit manipulation is a tool, not a pattern in the same sense as sliding window. Reach for it when you see: finding a unique/missing element, checking divisibility by 2, working with sets of boolean flags, or generating subsets. Pair it with XOR to eliminate duplicates.

5.8. Cyclic Sort

5.8.1. Intuition

Given an array containing integers in the range [1, n], the natural placement for the value `v` is at index `v - 1`. A *cyclic sort* exploits this: iterate through the array; if the current element is not at its correct index, swap it into place. Because each swap places at least one element correctly, the total number of swaps is at most n — giving an **O(n) in-place sort** for bounded integer arrays.

This makes it ideal for problems involving missing numbers, duplicate numbers, or finding all elements in the wrong place.

```
def cyclic_sort(nums):
    i = 0
    while i < len(nums):
        correct = nums[i] - 1               # where nums[i] should be
        if nums[i] != nums[correct]:        # not in place
            nums[i], nums[correct] = nums[correct], nums[i]
        else:
            i += 1                          # already correct, advance
    return nums
```

After sorting, a second pass finds all positions where `nums[i] != i + 1` — those are the missing or duplicate values.

```
def find_missing_numbers(nums):
    cyclic_sort(nums)
    return [i + 1 for i in range(len(nums)) if nums[i] != i + 1]
```

5.8.2. When to Apply

The problem involves an array of integers in the range [1, n] or [0, n], and asks for missing, duplicate, or corrupted values in O(n) time and O(1) space. The constraint "values are bounded by array length" is the key signal.

5.9. Arrays — Searching & Sorting

5.9.1. Binary Search

Binary search is the canonical divide-and-conquer algorithm for sorted arrays. The invariant is that the target, if present, lies within the current search interval `[lo, hi]`. Each step halves this interval:

$$T(n) = T(n/2) + O(1) \implies T(n) = O(\log n)$$

The most common implementation error is using `(lo + hi) // 2` when lo and hi are large integers — this can overflow in languages without arbitrary-precision integers. The safe form is `lo + (hi - lo) // 2`.

```
def binary_search(arr, target):
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = lo + (hi - lo) // 2
        if arr[mid] == target:   return mid
        elif arr[mid] < target:  lo = mid + 1
        else:                    hi = mid - 1
    return -1
```

Leftmost occurrence: When duplicates exist and you need the first index, use `hi = mid` (not `mid - 1`) when `arr[mid] >= target`. The loop runs until `lo == hi`, which is the answer.

Binary search on the answer: Many optimization problems have a monotone feasibility function — for some threshold X, all values X are infeasible and all values > X are feasible (or vice versa). Binary search on X directly without ever sorting the input.

5.9.2. Sorting Algorithms and Their Lineage

Each sorting algorithm was invented to address a limitation of its predecessor:

- **Insertion sort** is simple and O(n) on nearly-sorted data, but $O(n^2)$ in general.
- **Merge sort** achieves the O(n log n) lower bound with a stable sort, at the cost of O(n) extra space.
- **Quick sort** avoids the extra space of merge sort and is cache-friendly, but has a $O(n^2)$ worst case on sorted input without randomization.
- **Heap sort** achieves O(n log n) worst case *and* O(1) space, but is not stable and has poor cache behavior.
- **Counting / Radix sort** breaks the O(n log n) comparison-sort lower bound by exploiting the integer structure of keys — O(n) for bounded integers.

Algorithm	Best	Average	Worst	Space	Stable

Algorithm	Best	Average	Worst	Space	Stable
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes

The comparison-sort lower bound of $\Omega(n \log n)$ follows from the decision-tree model: a decision tree for sorting n elements has n! leaves, so its height is at least $\log_2(n!) = \Theta(n \log n)$ by Stirling's approximation.

5.10. Hash Tables

5.10.1. How Hash Tables Work

A hash table maps arbitrary keys to array indices via a hash function $h(k)$. The ideal hash function distributes keys uniformly, minimizing *collisions* — two keys mapping to the same index. Collisions are resolved by either chaining (linked list at each bucket) or open addressing (probing for the next open slot).

Load factor $\alpha = n/m$ (elements/buckets) governs performance. With chaining, expected lookup time is $O(1 + \alpha)$. Python's `dict` maintains $\alpha < 2/3$ by resizing, which keeps operations $O(1)$ in expectation.

5.10.2. The Hash Map as an Accelerator

The central use of hash maps in algorithms is to replace an O(n) search with an O(1) lookup. This unlocks O(n) solutions to problems that would otherwise be O(n^2).

Two Sum — the canonical example. For each element, check whether its complement has already been seen:

```
def two_sum(nums, target):
    seen = {}                        # value → index
    for i, n in enumerate(nums):
        complement = target - n
        if complement in seen:
            return [seen[complement], i]
        seen[n] = i
```

Anagram grouping — map each word to a canonical key (sorted letters) and group by key:

```
from collections import defaultdict

def group_anagrams(words):
    groups = defaultdict(list)
    for word in words:
        key = tuple(sorted(word))
        groups[key].append(word)
    return list(groups.values())
```

5.10.3. When to Apply

Any time a problem requires O(1) lookup by a computed key — frequency counting, caching previously computed values, detecting duplicates, or inverting a mapping. The signature phrase is "find X such that f(X) = target" — store computed values and look up what you need.

5.11. String Manipulation

5.11.1. Strings as Sliding Windows

Many string problems are window problems in disguise. The minimum window substring, longest substring without repeating characters, and find-all-anagrams problems all use the variable sliding window from Section 5.5.

5.11.2. Pattern Matching — KMP Algorithm

Naive pattern matching is O(nm). The Knuth-Morris-Pratt algorithm achieves O(n + m) by precomputing a *failure function* that tells us, when a mismatch occurs at position j in the pattern, how far back to reset j without re-examining already-matched characters.

The failure function $\pi[j]$ is the length of the longest proper prefix of `pattern[0..j]` that is also a suffix. This is precomputed in O(m):

```
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    length, i = 0, 1
    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
```

```
        elif length:
            length = lps[length - 1]    # don't increment i
        else:
            lps[i] = 0
            i += 1
    return lps
```

Searching then uses lps to skip redundant comparisons, running in O(n + m) total.

5.11.3. Palindromes — Expand Around Center

A palindrome reads the same forwards and backwards. To find the *longest palindromic substring*, the brute-force approach checks all $O(n^2)$ substrings in O(n) each — $O(n^3)$ total. Expanding around each center is $O(n^2)$ and O(1) space. Manacher's algorithm achieves O(n) using previously computed palindrome radii.

```
def longest_palindrome(s):
    best = ""
    for center in range(len(s)):
        for odd, even in [(center, center), (center, center + 1)]:
            lo, hi = odd, even
            while lo >= 0 and hi < len(s) and s[lo] == s[hi]:
                lo -= 1; hi += 1
            if hi - lo - 1 > len(best):
                best = s[lo + 1:hi]
    return best
```

5.12. Graphs

5.12.1. Representations

A graph $G = (V, E)$ can be stored as:

- **Adjacency list:** `{node: [neighbors]}` — O(V + E) space, O(degree) neighbor scan. Best for sparse graphs.
- **Adjacency matrix:** `matrix[i][j] = weight` — $O(V^2)$ space, O(1) edge lookup. Best for dense graphs or Floyd-Warshall.

Most interview graph problems use adjacency lists because real-world graphs are sparse.

5.12.2. BFS — Shortest Path in Unweighted Graphs

BFS explores vertices in *level order* — all vertices at distance 1, then distance 2, and so on. This guarantees that the first time a vertex is reached, it is reached via the shortest path. BFS is the correct algorithm when all edges have equal weight.

```
from collections import deque

def bfs_shortest(graph, source, target):
    queue = deque([(source, 0)])
    visited = {source}
    while queue:
        node, dist = queue.popleft()
        if node == target:
            return dist
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, dist + 1))
    return -1
```

Complexity: O(V + E) — each vertex and edge is processed at most once.

5.12.3. DFS — Connectivity, Cycles, Topological Order

DFS explores as deep as possible before backtracking. It is the foundation for: connected components, cycle detection, topological sort, and finding strongly connected components.

Topological sort (Kahn's algorithm — BFS-based) linearizes a DAG so that for every directed edge `u → v`, u appears before v. It works by repeatedly removing vertices with in-degree zero:

```
from collections import deque, defaultdict

def topological_sort(n, edges):
    graph = defaultdict(list)
    indegree = [0] * n
    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1
    queue = deque(node for node in range(n) if indegree[node] == 0)
    order = []
    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
```

```
                queue.append(neighbor)
    return order if len(order) == n else []     # empty list means a cycle exists
```

5.12.4. Dijkstra — Shortest Path with Non-negative Weights

Dijkstra's algorithm generalizes BFS to weighted graphs by using a *priority queue* instead of a plain queue. The priority queue always expands the vertex with the smallest known distance — a greedy choice that is correct when all edge weights are non-negative.

Each vertex is finalized at most once. With a binary heap, the complexity is $O((V + E) \log V)$.

```
import heapq

def dijkstra(graph, source):
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    heap = [(0, source)]
    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]:
            continue                        # stale entry - discard
        for v, weight in graph[u]:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                heapq.heappush(heap, (dist[v], v))
    return dist
```

Why Dijkstra fails on negative edges: the greedy finalization is incorrect when a later path through a negative edge could produce a shorter distance to an already-finalized vertex. Use Bellman-Ford (O(VE)) in that case.

5.13. Trees

5.13.1. Tree Traversals and Their Meaning

A binary tree has three natural recursive traversals. The choice of traversal is determined by *when you need to process the current node relative to its children*:

- **Preorder (root → left → right):** used when you need to process a node before knowing about its subtrees — serialization, cloning a tree, printing a directory structure.
- **Inorder (left → root → right):** for a BST, produces elements in sorted order — this is the defining property of BSTs.

- **Postorder (left → right → root):** used when a node's computation depends on its children — computing subtree sizes, deleting a tree, evaluating an expression tree.
- **Level-order (BFS):** used when you need to process nodes level by level — finding the minimum depth, connecting nodes at the same level, right-side view.

5.13.2. Recursive Tree Thinking

Most tree problems decompose into: solve for left subtree, solve for right subtree, combine. Define clearly what your function *returns* and what it *does as a side effect*.

```
def max_path_sum(root):
    """
    Returns the maximum gain from root downward (for parent's use).
    Updates a global maximum as a side effect.
    """
    best = [float('-inf')]

    def dfs(node):
        if not node: return 0
        left  = max(dfs(node.left),  0)    # ignore negative branches
        right = max(dfs(node.right), 0)
        best[0] = max(best[0], node.val + left + right)   # path through node
        return node.val + max(left, right)                 # single branch upward

    dfs(root)
    return best[0]
```

5.13.3. Binary Search Trees

A BST maintains the invariant that all keys in the left subtree are strictly less than the root's key, and all keys in the right subtree are strictly greater. This invariant enables O(log n) search, insert, and delete on a *balanced* BST.

A BST degenerates to a linked list (O(n) operations) when elements are inserted in sorted order. Self-balancing BSTs (AVL, Red-Black) prevent this by maintaining a height invariant after every operation.

Validating a BST requires propagating valid ranges, not just comparing a node to its immediate children:

```
def is_valid_bst(root, lo=float('-inf'), hi=float('inf')):
    if not root: return True
    if not (lo < root.val < hi): return False
    return (is_valid_bst(root.left,  lo,       root.val) and
            is_valid_bst(root.right, root.val, hi))
```

5.14. Stacks & Queues

5.14.1. The Monotonic Stack

A plain stack tracks the history of values. A *monotonic* stack maintains an additional invariant — the stack is always sorted (increasing or decreasing). When a new element violates the invariant, we pop until it is restored.

This is the standard tool for the family of "next greater / smaller element" problems. Each element is pushed and popped at most once, so the total work is O(n).

```
def next_greater_element(arr):
    """For each element, find the first element to its right that is larger."""
    n = len(arr)
    result = [-1] * n
    stack = []                      # stores indices; invariant: arr[stack] is decreasing
    for i in range(n):
        while stack and arr[stack[-1]] < arr[i]:
            result[stack.pop()] = arr[i]
        stack.append(i)
    return result
```

Largest rectangle in histogram extends this idea: for each bar, find the first shorter bar to its left and right — the current bar is the height of the largest rectangle spanning that range. Two monotonic stack passes (or one combined pass) solve it in O(n).

5.14.2. Queues for Sliding Window Maximum

The sliding window maximum problem — find the maximum in each window of size k — appears in image processing, financial time series, and streaming analytics. A min-heap gives O(n log k); a *monotonic deque* gives O(n) by discarding elements that can never be the maximum of any future window.

```
from collections import deque

def sliding_window_maximum(nums, k):
    dq = deque()        # stores indices; invariant: nums[dq] is decreasing
    result = []
    for i, n in enumerate(nums):
        while dq and nums[dq[-1]] < n:
            dq.pop()                            # smaller elements can never be max
        dq.append(i)
        if dq[0] == i - k:
            dq.popleft()                        # front is out of window
        if i >= k - 1:
            result.append(nums[dq[0]])          # front is the window maximum
    return result
```

5.15. Linked Lists

5.15.1. The Dummy Node Technique

Edge cases in linked list problems almost always involve the head: deleting the head, inserting before the head, or an empty list. The *dummy node* technique introduces a sentinel before the head, allowing uniform treatment of all nodes — the head is now just another non-first node from the dummy's perspective.

```
def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    fast = slow = dummy
    for _ in range(n + 1):      # advance fast n+1 steps ahead
        fast = fast.next
    while fast:                 # move both until fast reaches the end
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next  # remove the nth node from end
    return dummy.next
```

5.15.2. Reversing a Linked List

Reversal is the foundation for many list problems. The iterative version uses three pointers — prev, curr, next — and runs in O(n) with O(1) space.

```
def reverse_list(head):
    prev, curr = None, head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev, curr = curr, nxt
    return prev
```

Reversing a sublist (from position `m` to `n`) follows the same logic but requires careful pointer bookkeeping at the boundaries — the dummy node again simplifies the head case.

5.16. Heaps

5.16.1. The Heap Property and Its Consequences

A binary heap is a *complete* binary tree satisfying the heap property: for a min-heap, every parent is its children. Stored as an array, the parent of node `i` is at `(i-1)//2`, and children are at `2i+1` and `2i+2`.

The heap property enables: - **O(1) access to the minimum** (root). - **O(log n) insertion** (append, then sift up). - **O(log n) deletion of minimum** (swap root with last, remove last, sift root down). - **O(n) heapification** from an unordered array — not O(n log n) as might be expected, because lower nodes need less sifting. The analysis uses the geometric series $\sum_{h=0}^{\log n} n/2^h \cdot h = O(n)$.

5.16.2. Top-K Problems

The heap is the canonical data structure for *streaming top-k*: maintain a min-heap of size k; for each new element, if it is larger than the heap minimum, replace the minimum. After all elements, the heap contains the k largest.

```
import heapq

def k_largest(stream, k):
    heap = []
    for x in stream:
        heapq.heappush(heap, x)
        if len(heap) > k:
            heapq.heappop(heap)             # discard the smallest
    return sorted(heap, reverse=True)       # k largest elements
```

This is O(n log k) time and O(k) space — far better than sorting all n elements when k n.

5.16.3. K-way Merge

Merging k sorted lists appears in external sorting, distributed systems, and database query processing. A min-heap tracks the current minimum across all k list heads. Each extraction takes O(log k), and there are n total elements, giving **O(n log k)**.

5.17. Recursion & Backtracking

5.17.1. The Recursion Mindset

Recursion is not a trick — it is a mathematical induction made executable. Every recursive solution embeds a proof: *assume the function works correctly on smaller inputs; show it works on the current input using those results.*

The structure of any recursive solution is:

1. **Base case:** a problem small enough to solve directly.
2. **Recursive step:** break the problem into smaller subproblems, solve them, and combine.

The call stack stores the state of each recursive invocation. A recursion of depth d uses O(d) stack space — this is the "hidden" space cost of recursive algorithms.

5.17.2. Backtracking

Backtracking is *systematic exploration with pruning*. It builds a solution incrementally, abandoning (pruning) partial solutions as soon as they are guaranteed to not lead to a valid complete solution.

The generic template:

```
def backtrack(state, choices):
    if is_complete(state):
        record(state)
        return
    for choice in choices:
        if is_valid(state, choice):
            apply(state, choice)
            backtrack(state, remaining_choices(choices, choice))
            undo(state, choice)             # restore state
```

The `undo` step is what makes this *backtracking* rather than a greedy algorithm — we explore one branch fully, then restore the state and try another.

5.17.3. N-Queens

Place n queens on an n×n chessboard such that no two queens attack each other. The key insight is that attacks are determined by column, and two diagonals. Tracking these three sets makes the validity check O(1).

```
def solve_n_queens(n):
    cols = set()
    diag1 = set()       # row - col is constant along a diagonal
    diag2 = set()       # row + col is constant along the other diagonal
    result = []
```

```
    def backtrack(row, board):
        if row == n:
            result.append(["".join(r) for r in board])
            return
        for col in range(n):
            if col in cols or (row - col) in diag1 or (row + col) in diag2:
                continue
            cols.add(col); diag1.add(row - col); diag2.add(row + col)
            board[row][col] = 'Q'
            backtrack(row + 1, board)
            board[row][col] = '.'
            cols.discard(col); diag1.discard(row - col); diag2.discard(row + col)

    backtrack(0, [['.']*n for _ in range(n)])
    return result
```

Complexity: O(n!) in the worst case — there are n choices for the first queen, n-1 for the second (after pruning), and so on. In practice, pruning reduces this dramatically.

5.18. Dynamic Programming

5.18.1. The Two Conditions

A problem is amenable to dynamic programming if and only if it has:

1. **Optimal substructure:** an optimal solution to the problem contains optimal solutions to its subproblems.
2. **Overlapping subproblems:** the same subproblems are solved multiple times in a naive recursive solution.

If the subproblems are *independent* (non-overlapping), divide-and-conquer is the right tool, not DP. If the problem has optimal substructure but no overlapping subproblems, greedy may apply.

5.18.2. Top-Down vs. Bottom-Up

Top-down (memoization): write the recursive solution naturally, then cache results. Intuitive but uses O(n) call-stack space.

Bottom-up (tabulation): identify the dependency order of subproblems and fill a table iteratively. Eliminates recursion overhead and often allows space optimization.

The two approaches have identical time complexity but different constants. Bottom-up is generally preferred in production.

5.18.3. The DP Design Process

For every DP problem, explicitly answer four questions before writing code:

1. **State definition:** what is `dp[i]` (or `dp[i][j]`)? Write it as a sentence.
2. **Transition:** how does `dp[i]` relate to `dp[i-1]` (or other previous states)?
3. **Base case:** what are the smallest subproblems, and what are their solutions?
4. **Answer:** which entry in the table is the final answer?

5.18.4. Canonical Examples

Coin Change — unbounded knapsack. `dp[a]` = minimum coins needed to make amount `a`. Transition: for each coin `c`, `dp[a] = min(dp[a], dp[a-c] + 1)`.

```
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0                              # base: 0 coins to make amount 0
    for a in range(1, amount + 1):
        for c in coins:
            if c <= a:
                dp[a] = min(dp[a], dp[a - c] + 1)
    return dp[amount] if dp[amount] != float('inf') else -1
```

Longest Common Subsequence — `dp[i][j]` = LCS length of `s1[:i]` and `s2[:j]`. The transition branches on whether `s1[i-1] == s2[j-1]`.

```
def lcs(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]
```

Longest Increasing Subsequence (O(n log n)) — the naive DP is $O(n^2)$. The optimized version maintains a *tails* array where `tails[i]` is the smallest tail element of any increasing subsequence of length `i+1`. Binary search on `tails` gives O(n log n).

```
from bisect import bisect_left

def lis(nums):
    tails = []
    for n in nums:
```

```
        pos = bisect_left(tails, n)
        if pos == len(tails): tails.append(n)
        else:                 tails[pos] = n
    return len(tails)
```

The `tails` array is not the actual LIS, but its *length* is correct. To reconstruct the actual subsequence, track parent pointers.

Interview Focus

Pattern recognition: given a new problem, state which pattern applies and *why* before writing any code. **Complexity derivation:** explain why sliding window is O(n) — each element enters and leaves the window at most once. **DP design:** for any DP problem, state your `dp[i]` definition in words, write the transition, and identify the base case before coding. **Tradeoffs:** when would you choose Bellman-Ford over Dijkstra? When does memoization beat tabulation? Know not just *what* each algorithm does but *when* it is the right choice and *why* alternatives fail.

6. ML Data Pipelines

Objectives

After this chapter, you should understand why each preprocessing decision exists — not just what to do, but what goes wrong when you do it incorrectly. You should be able to design a complete training pipeline for tabular, text, vision, and time-series data, and explain the pipeline differences a research scientist would encounter across these modalities.

Reading

- *Designing Machine Learning Systems* — Chip Huyen, Chapters 4–7
- *Feature Engineering for Machine Learning* — Zheng & Casari
- *Speech and Language Processing* — Jurafsky & Martin, Chapters 2–3
- *Programming PyTorch for Deep Learning* — Ian Pointer

6.1. What a Pipeline Is and Why It Matters

A data pipeline is the sequence of transformations that converts raw data into the numerical representations a model can consume, and then delivers those representations consistently at both training time and serving time. The word *consistently* is load-bearing: a pipeline that applies different transformations at training versus serving time — even subtly — will produce a model that fails in production despite strong offline metrics. This failure mode is called **training-serving skew**, and it is among the most common production ML bugs.

Every pipeline, regardless of modality, passes through the same logical stages:

```
Raw Data → Cleaning → Representation → Normalization → Augmentation
         → Splitting → Model Training → Evaluation → Serving
```

What differs across modalities is *what* each stage does and *why*. Text requires tokenization; images require spatial augmentation; tabular data requires categorical encoding. The mathematical reasons behind each choice are what this chapter covers.

A second cross-cutting concern is **data leakage** — the accidental inclusion of information in the training set that would not be available at prediction time. Leakage causes inflated training metrics that collapse at deployment. Every splitting and preprocessing decision must be evaluated through the lens of "would I have this information when making a real prediction?"

6.2. Tabular Data Pipelines

6.2.1. The Nature of Tabular Data

Tabular data is the most common format in industry: rows are observations, columns are features. What makes tabular data challenging is *heterogeneity* — a single table may contain integers, floats, free-text strings, categorical labels, timestamps, and boolean flags, each requiring a different preprocessing treatment.

Unlike images or text, tabular data has no natural spatial or sequential structure that a model can exploit. Every feature must be made meaningful by the pipeline designer. This places more burden on feature engineering here than in any other modality.

6.2.2. Numerical Features: Scaling

Most learning algorithms — linear models, SVMs, neural networks, and k-NN — are sensitive to the *scale* of numerical features. A feature measured in thousands (income) will dominate a feature measured in ones (number of children) in any distance or gradient computation. Scaling removes this accidental dominance.

Three scalers address three different distributional situations:

StandardScaler centers each feature at zero and scales to unit variance:

$$x' = \frac{x - \mu}{\sigma}$$

This is the default choice for features that are approximately Gaussian. It preserves the shape of the distribution while making all features commensurable. Neural networks train faster when inputs are zero-mean and unit-variance because gradients flow more uniformly.

MinMaxScaler maps features to $[0, 1]$:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

It is appropriate when the feature has a bounded natural range (e.g., image pixel values 0–255, probabilities, percentages). Its weakness is sensitivity to outliers: a single extreme value compresses all other values toward zero.

RobustScaler uses the median and interquartile range instead of the mean and standard deviation:

$$x' = \frac{x - \text{median}(x)}{IQR(x)}$$

Since the median and IQR are resistant to outliers by construction, RobustScaler is the correct choice for skewed distributions and tabular data from the real world, where outliers are the norm rather than the exception.

Tree-based models (decision trees, random forests, gradient boosting) are invariant to monotone transformations of individual features — they never compute distances or gradients across features — so scaling has no effect on them.

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.pipeline import Pipeline

# The pipeline enforces fit-on-train, transform-on-test - eliminates leakage
pipe = Pipeline([
    ("scaler", RobustScaler()),
    ("model",  LogisticRegression()),
])
pipe.fit(X_train, y_train)
pipe.score(X_test, y_test)       # scaler never sees X_test statistics
```

The `Pipeline` object is not merely a convenience — it is a correctness guarantee. Fitting the scaler before splitting the data is one of the most common leakage mistakes.

6.2.3. Categorical Features: Encoding

A categorical variable takes one of a finite set of discrete values. Models require numerical inputs, so categories must be encoded. The choice of encoding determines whether the model can learn the right relationships.

One-hot encoding creates a binary indicator column for each category. It imposes no ordering — the model treats each category as independent. This is correct for *nominal* variables (city, product type, color). Its cost is dimensionality: a feature with k categories becomes k binary columns. For high-cardinality features (zip codes, user IDs, product SKUs), this becomes prohibitive.

Ordinal encoding assigns integers in order: `low=0, medium=1, high=2`. It is correct for *ordinal* variables where the order is meaningful. Applying it to nominal variables is wrong — it tells the model that "Paris > London > Tokyo" in a way that has no semantic content.

Target encoding replaces each category with the mean of the target variable conditioned on that category:

$$\text{enc}(c) = \mathbb{E}[y \mid x = c]$$

This is powerful for high-cardinality nominal features and is what many Kaggle-winning solutions use. Its critical weakness is **leakage**: computing the encoding on the full training set before cross-validation allows the model to see target information that would not be available for new, unseen categories. The correct implementation computes encodings *within* each cross-validation fold.

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import TargetEncoder

# OneHotEncoder with unknown category handling
ohe = OneHotEncoder(sparse_output=False, handle_unknown="ignore")

# TargetEncoder with cross-fitting to prevent leakage
te = TargetEncoder(cv=5, smooth="auto")
```

6.2.4. Missing Values: A Statistical Framework

Missing data is not random noise to discard — it carries information about the data-generating process. Before imputing, identify *why* values are missing:

- **MCAR (Missing Completely at Random):** missingness is independent of all variables. Simple imputation is safe.
- **MAR (Missing at Random):** missingness depends on observed variables but not on the missing value itself. Imputation conditioned on observed covariates is appropriate.
- **MNAR (Missing Not at Random):** missingness depends on the unobserved value itself (e.g., high-earners skip the income field). No imputation strategy is unbiased — the missingness pattern must be modeled.

For MNAR variables, adding a binary indicator column before imputing is essential. The indicator preserves the information that the value was missing, which the model can then use as a signal.

```
from sklearn.impute import SimpleImputer, KNNImputer

# For MNAR variables: add indicator before imputing
X["income_missing"] = X["income"].isna().astype(int)
X["income"] = X["income"].fillna(X["income"].median())

# KNN imputation: imputes using the k nearest complete observations
# Appropriate when features are correlated and MCAR/MAR holds
knn_imp = KNNImputer(n_neighbors=5)
```

6.2.5. Feature Engineering for Tabular Data

Feature engineering is the highest-leverage activity in tabular ML. A domain-informed feature often matters more than a better model. The guiding principle is to encode domain knowledge into the representation so the model does not have to discover it from scratch.

Common transformations:

- **Log transform** for right-skewed distributions (income, counts, prices). $\log(1 + x)$ handles zeros. This maps a multiplicative relationship to an additive one, which linear models can capture.

- **Polynomial and interaction features** for capturing nonlinearities. A model that cannot learn $x_1 \cdot x_2$ from x_1 and x_2 separately benefits from seeing the product as a feature.
- **Binning** converts a continuous variable to an ordinal category, which can help when the relationship is non-monotone (e.g., age vs. income has a hump shape, not a line).
- **Ratio features** encode domain knowledge directly: debt-to-income, click-through rate, conversion rate.

```
import numpy as np
import pandas as pd

df["log_price"]         = np.log1p(df["price"])
df["debt_to_income"]    = df["debt"] / (df["income"] + 1e-6)
df["age_x_experience"] = df["age"] * df["years_experience"]
df["is_weekend"]        = df["timestamp"].dt.dayofweek.isin([5, 6]).astype(int)
```

6.2.6. Data Leakage

Leakage is the single most important concept in pipeline design. It causes a model to perform well in offline evaluation and fail in production — a failure that is both common and expensive.

Leakage Type	Mechanism	Prevention
Target leakage	Feature derived from or correlated with target via post-event information	Audit feature creation timestamps relative to label timestamp
Train-test contamination	Scaler or imputer fit on the full dataset before splitting	Always split before fitting any preprocessor
Group leakage	Same entity (patient, user, product) appears in both train and test	Use `GroupKFold`; split by entity, not by row
Temporal leakage	Future information used to predict the past	Use `TimeSeriesSplit`; enforce strict chronological splits
Preprocessing leakage	Target encoding computed across full dataset	Compute encodings within cross-validation folds

6.2.7. Class Imbalance

In many real-world problems — fraud detection, medical diagnosis, rare event prediction — the positive class is a small fraction of the data. A model that predicts the majority class for every instance achieves high accuracy but zero utility. The appropriate response depends on the degree of imbalance and the cost asymmetry between false positives and false negatives.

Class weights modify the loss function to penalize mistakes on the minority class more heavily. This is the lowest-overhead fix, built into most sklearn estimators, and should always be tried first.

SMOTE (Synthetic Minority Oversampling Technique) generates new minority-class samples by interpolating between existing ones in feature space. For a minority sample x_i and one of its k nearest minority neighbors x_j:

$$x_{\text{new}} = x_i + \lambda(x_j - x_i), \quad \lambda \sim \text{Uniform}(0, 1)$$

This is appropriate for tabular data with continuous features. It fails when the minority class forms a complex non-convex manifold or when features are highly categorical.

Threshold tuning is the most underused tool: after training, choose the classification threshold to maximize the operating metric (F1, recall at fixed precision, or a business cost function). The threshold need not be 0.5.

```
from sklearn.metrics import precision_recall_curve

prec, rec, thresholds = precision_recall_curve(y_val, y_scores)
f1 = 2 * prec * rec / (prec + rec + 1e-9)
best_threshold = thresholds[f1[:-1].argmax()]
y_pred = (y_scores >= best_threshold).astype(int)
```

6.3. NLP Data Pipelines

6.3.1. From Characters to Meaning: The Representation Problem

Text is a sequence of discrete symbols. Every NLP pipeline must solve the *representation problem*: how do you convert a string of characters into a fixed numerical representation that captures semantic and syntactic meaning?

The history of NLP pipeline design is the history of increasingly sophisticated answers to this question:

```
Characters → Words (count-based) → Words (distributional, Word2Vec)
          → Subwords (BPE) → Contextual embeddings (BERT) → LLM tokenization
```

Each transition was driven by a concrete failure of the previous approach.

6.3.2. Classical Text Preprocessing

Before neural methods, text required explicit cleaning to remove noise that bag-of-words models would otherwise treat as signal.

```
import re
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r"[^a-z0-9\s]", "", text)     # remove punctuation
    tokens = text.split()
    stop_words = set(stopwords.words("english"))
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(t) for t in tokens if t not in stop_words]
    return tokens
```

Whether to keep stopwords, apply stemming vs. lemmatization, or remove punctuation depends on the downstream task. For sentiment analysis, "not" is crucial — removing it as a stopword is wrong. For topic modeling, stopwords add noise. These decisions require understanding the task, not just following a recipe.

6.3.3. Bag of Words and TF-IDF

Bag of Words (BoW) represents a document as a vector of word counts over a fixed vocabulary. It loses word order entirely but captures topic content. Two documents about "machine learning" will have similar BoW vectors regardless of sentence structure.

TF-IDF reweights BoW by penalizing words that appear in many documents (like "the") and rewarding words that are discriminative (appear frequently in a few documents but rarely overall):

$$\text{TF-IDF}(t, d) = \underbrace{\frac{f_{t,d}}{\sum_{t'} f_{t',d}}}_{\text{term frequency}} \times \underbrace{\log \frac{N}{|\{d : t \in d\}|}}_{\text{inverse document frequency}}$$

The IDF term is the information-theoretic insight: a word that appears in every document has zero discriminative power — its presence tells you nothing about document identity.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer(
    max_features=50_000,
    ngram_range=(1, 2),          # unigrams and bigrams
    min_df=2,                    # ignore very rare terms
    max_df=0.95,                 # ignore near-universal terms
    sublinear_tf=True,           # use log(1 + tf) to dampen high counts
)
X_train_tfidf = tfidf.fit_transform(train_texts)
X_test_tfidf  = tfidf.transform(test_texts)      # never refit on test
```

6.3.4. Tokenization: The Vocabulary Problem

Word-level tokenization fails in two ways. First, the vocabulary must be fixed at training time — any word not seen during training becomes out-of-vocabulary (OOV). Second, morphologically related words ("run", "running", "ran") are treated as unrelated.

Byte Pair Encoding (BPE), used by GPT models, solves both problems by operating at the subword level. It starts from a character vocabulary and iteratively merges the most frequent adjacent pair of symbols. After enough merges, common words become single tokens, while rare words are decomposed into meaningful subword units. "unhappiness" might tokenize as `["un", "happiness"]` — both are meaningful subwords, and neither is OOV.

The training procedure:

1. Initialize vocabulary as all individual characters plus a special end-of-word symbol.
2. Count all adjacent symbol pairs in the corpus.
3. Merge the most frequent pair into a new symbol.
4. Repeat until the vocabulary reaches the target size.

WordPiece (used by BERT) is similar but selects merges to maximize the likelihood of the training corpus under a language model rather than by raw frequency.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
tokens = tokenizer(
    "The quick brown fox",
    padding="max_length",
    truncation=True,
    max_length=128,
    return_tensors="pt"
)
# tokens: {"input_ids": ..., "attention_mask": ..., "token_type_ids": ...}
```

The **attention mask** is as important as the token IDs: it tells the model which positions are real tokens and which are padding, preventing the padding from influencing the attention computation.

6.3.5. Word Embeddings: Distributional Semantics

Word2Vec (2013) operationalized the *distributional hypothesis*: words that appear in similar contexts have similar meanings. It trains a shallow neural network on one of two tasks:

- **CBOW (Continuous Bag of Words):** predict the center word from its context.
- **Skip-gram:** predict context words from the center word.

After training, the weight matrix is the embedding matrix — each row is a dense vector representation of a word. The famous "king − man + woman queen" result emerges from the geometry of this space, not from any explicit encoding of gender relationships.

Word2Vec's limitation is that each word has a single embedding regardless of context. "Bank" has the same vector whether it refers to a river bank or a financial institution. **Contextual embeddings** (ELMo, BERT, GPT) produce a different vector for each occurrence of a word, conditioned on the full sentence. This is the fundamental shift that drove the modern NLP revolution.

6.3.6. Text Data Augmentation

Augmenting text while preserving labels is harder than augmenting images, because small changes can flip the meaning ("I do not like this" → "I like this" after removing "not").

Technique	Method	Preserves Label?	Notes
Synonym replacement	Replace n random words with synonyms	Usually	Avoid negations, sentiment words
Random insertion	Insert a random synonym at a random position	Usually	Mild distributional shift
Back-translation	Translate to language X, then back	Usually	Creates paraphrases
EDA (Easy Data Augmentation)	Swap, delete, insert, replace	Usually	Effective for low-data regimes
Mixup for text	Interpolate embeddings + labels	By construction	Operates in embedding space

```
import nlpaug.augmenter.word as naw

aug = naw.SynonymAug(aug_src="wordnet", aug_p=0.1)   # replace 10% of words
augmented = aug.augment("The model achieved state of the art results")
```

6.3.7. Handling Variable-Length Sequences

Models require fixed-size inputs. Text sequences vary in length. Two strategies:

Padding and truncation: pad shorter sequences to a fixed maximum length with a special `[PAD]` token; truncate longer sequences. This is simple but wastes computation on padding tokens. The attention mask prevents padding from contributing to attention scores.

Dynamic batching with bucketing: group sequences of similar length into the same batch, minimizing padding within each batch. Sort the dataset by length, then take batches of consecutive sequences. This reduces wasted computation by 20–40% on typical datasets.

```python
from torch.nn.utils.rnn import pad_sequence

def collate_fn(batch):
    texts, labels = zip(*batch)
    # pad_sequence expects list of tensors, pads to longest in batch
    padded = pad_sequence(texts, batch_first=True, padding_value=0)
    return padded, torch.tensor(labels)
```

6.4. Computer Vision Pipelines

6.4.1. Images as Tensors

A digital image is a 3-dimensional tensor of shape `[C, H, W]` — channels × height × width. For RGB images, C=3. Each element is a pixel intensity, typically an integer in [0, 255] or a float in [0.0, 1.0] after normalization.

The core challenge in vision pipelines is that real-world image datasets exhibit enormous variation: objects appear at different scales, orientations, lighting conditions, and positions. A model trained on tightly cropped, evenly lit studio photographs will fail on natural scene photographs. The pipeline must bridge this gap through normalization and augmentation.

6.4.2. Normalization

The first transformation applied to any image in a deep learning pipeline is normalization. Dividing by 255 converts integer pixels to [0, 1]. Subtracting the dataset mean and dividing by the dataset standard deviation, per channel, achieves zero-mean unit-variance inputs:

$$x'_c = \frac{x_c - \mu_c}{\sigma_c}$$

For models pretrained on ImageNet, the canonical statistics are used regardless of the target dataset, because the model's weights were tuned for inputs in this distribution:

$$\mu = [0.485,\ 0.456,\ 0.406], \quad \sigma = [0.229,\ 0.224,\ 0.225]$$

Deviating from these statistics when fine-tuning from an ImageNet-pretrained checkpoint will cause the first layer to receive out-of-distribution inputs, slowing convergence.

6.4.3. Data Augmentation for Vision

Augmentation is the primary tool for reducing overfitting in vision models. It creates new training examples by applying transformations that preserve the semantic label while altering the pixel distribution. The transformations are applied randomly during training but never during validation or testing.

Geometric transformations alter spatial structure:

- Random horizontal flip: valid for most natural scene categories, but not for tasks where orientation matters (digit recognition, text recognition).
- Random crop: forces the model to recognize objects from partial views, improving robustness to occlusion and translation.
- Random rotation: improves rotational invariance where appropriate.

Photometric transformations alter appearance without changing geometry:

- Color jitter: randomly adjust brightness, contrast, saturation, and hue. Forces the model to rely on shape, not color, for classification.
- Gaussian blur: simulates depth-of-field variation.
- Grayscale conversion: removes color information, forcing reliance on texture and shape.

Advanced augmentations mix samples:

MixUp interpolates two images and their one-hot labels simultaneously:

$$\tilde{x} = \lambda x_i + (1-\lambda)x_j, \quad \tilde{y} = \lambda y_i + (1-\lambda)y_j, \quad \lambda \sim \text{Beta}(\alpha, \alpha)$$

This forces the model to predict proportional probabilities for mixed images, acting as a regularizer and improving calibration.

CutMix replaces a rectangular patch of one image with a patch from another, mixing labels proportionally to the patch area. It is stronger than MixUp for recognition tasks because it preserves local texture statistics within each patch.

```
import torchvision.transforms as T
from torchvision.transforms import v2

# Training pipeline with standard augmentations
train_transform = T.Compose([
    T.RandomResizedCrop(224, scale=(0.08, 1.0)),    # crop to 224×224
    T.RandomHorizontalFlip(p=0.5),
    T.ColorJitter(brightness=0.4, contrast=0.4,
                  saturation=0.4, hue=0.1),
    T.RandomGrayscale(p=0.2),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]),
])
```

```
# Validation pipeline: no augmentation, only deterministic resizing
val_transform = T.Compose([
    T.Resize(256),
    T.CenterCrop(224),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]),
])
```

The asymmetry between training and validation transforms is fundamental: augmentation improves training robustness but must never be applied at evaluation time, because it would make results non-reproducible and introduce noise into the metric.

6.4.4. Efficient Data Loading

In vision training, the GPU is often idle waiting for data. The DataLoader pipeline must feed data fast enough to keep GPU utilization above 90%.

```
from torch.utils.data import DataLoader

loader = DataLoader(
    dataset,
    batch_size=256,
    shuffle=True,
    num_workers=8,          # parallel CPU workers for decoding and augmentation
    pin_memory=True,        # page-locked memory for faster CPU→GPU transfer
    prefetch_factor=2,      # each worker prefetches 2 batches ahead
    persistent_workers=True,   # workers stay alive between epochs
)
```

Bottleneck diagnosis: if GPU utilization is low during training, the bottleneck is data loading. Solutions: increase `num_workers`, move to faster storage (SSD over NFS), use DALI (NVIDIA Data Loading Library) for GPU-accelerated decoding, or pre-cache decoded images in RAM.

6.4.5. Transfer Learning and the ImageNet Bottleneck

Almost all modern vision pipelines start with a pretrained backbone. The ImageNet-pretrained ResNet or ViT has already learned general visual features — edges, textures, shapes, object parts — that transfer across tasks. Fine-tuning on a new dataset typically requires only 10–100× fewer labeled examples than training from scratch.

Two fine-tuning strategies:

Feature extraction: freeze all backbone weights, train only the classification head. Appropriate when the target dataset is small and similar to the pretraining distribution. The backbone becomes a fixed feature extractor.

Full fine-tuning: unfreeze all weights and train end-to-end with a small learning rate. Appropriate when the target dataset is large or dissimilar from the pretraining distribution. Use a lower learning rate for the backbone than the head to avoid destroying pretrained features.

```
import torchvision.models as models

backbone = models.resnet50(weights="IMAGENET1K_V2")

# Feature extraction: freeze backbone
for param in backbone.parameters():
    param.requires_grad = False

# Replace final layer for new number of classes
backbone.fc = nn.Linear(backbone.fc.in_features, num_classes)

# Fine-tuning: different learning rates
optimizer = torch.optim.Adam([
    {"params": backbone.layer4.parameters(), "lr": 1e-4},
    {"params": backbone.fc.parameters(),     "lr": 1e-3},
])
```

6.5. Time Series Pipelines

6.5.1. Why Time Series Requires Special Treatment

Time series data violates the independence assumption that underlies standard ML pipelines. In a tabular dataset, the order of rows is arbitrary — you can shuffle them freely. In a time series, the order *is* the information. Shuffling destroys it.

This has cascading consequences for every pipeline stage:

- **Splitting must be chronological.** A random train/test split would allow future data to appear in the training set, producing a leaky model that has memorized the future.
- **Features must be computed from the past only.** A rolling mean computed over a centered window uses future data. It must be backward-looking.
- **Cross-validation must respect time.** Standard k-fold randomly assigns data to folds — forbidden here. Walk-forward validation is required.

6.5.2. Stationarity and Transformations

A time series is *weakly stationary* if its mean and autocovariance do not depend on time. Most statistical models and many ML models assume stationarity. Non-stationary series (those with trends, changing variance, or structural breaks) must be transformed.

Differencing removes trends by replacing each value with the change from the previous value:

$$\Delta Y_t = Y_t - Y_{t-1}$$

A series with a linear trend becomes stationary after first differencing. A series with a quadratic trend requires second differencing. Differencing is the appropriate transformation when the non-stationarity is *stochastic* (a random walk). For a *deterministic* trend (a polynomial in time), subtracting the fitted trend is more efficient.

Log transformation stabilizes variance in series where variability grows with the level — common in economic and financial data. $\log(Y_t)$ converts multiplicative dynamics to additive ones.

```
import pandas as pd
import numpy as np

# First-order differencing
df["returns"]      = df["price"].pct_change()         # proportional change
df["log_price"]    = np.log(df["price"])
df["diff1"]        = df["price"].diff(1)              # absolute change
df["diff_log"]     = df["log_price"].diff(1)          # log returns
```

6.5.3. Feature Engineering for Time Series

Unlike tabular data, where features are given, time series features must be *constructed* from the historical record. The art is choosing window sizes and aggregations that capture the relevant temporal dynamics.

Lag features give the model direct access to past values. A lag-1 feature is Y_{t-1}, a lag-7 feature is Y_{t-7} (one week ago for daily data). The choice of lags should be informed by the autocorrelation structure of the series.

Rolling statistics summarize recent history:

- Rolling mean: captures the local level.
- Rolling standard deviation: captures local volatility.
- Rolling min/max: captures recent extremes.

Calendar features capture seasonality and periodic patterns:

- Hour, day of week, month, quarter — for diurnal and seasonal patterns.
- Is-holiday, days-since-last-holiday — for event-driven dynamics.
- Fourier features: $\sin(2\pi kt/P)$ and $\cos(2\pi kt/P)$ for period P, encoding smooth periodicity.

```
def build_time_features(df, target_col, lags, windows):
    for lag in lags:
        df[f"lag_{lag}"] = df[target_col].shift(lag)
    for w in windows:
        df[f"roll_mean_{w}"] = df[target_col].shift(1).rolling(w).mean()
        df[f"roll_std_{w}"]  = df[target_col].shift(1).rolling(w).std()
    df["hour"]        = df.index.hour
```

```
    df["dayofweek"]   = df.index.dayofweek
    df["month"]       = df.index.month
    for k in [1, 2, 3]:
        df[f"sin_{k}"] = np.sin(2 * np.pi * k * df.index.dayofyear / 365)
        df[f"cos_{k}"] = np.cos(2 * np.pi * k * df.index.dayofyear / 365)
    df.dropna(inplace=True)     # lags create NaNs at the start
    return df
```

The `.shift(1)` on rolling features is critical: without it, the rolling mean at time t includes Y_t itself, which would be a future data leak.

6.5.4. Walk-Forward Validation

Standard k-fold cross-validation is incorrect for time series because it allows future data to appear in training folds. Walk-forward (also called *expanding window*) validation maintains temporal order: the training set always ends before the validation set begins.

```
Fold 1: Train [1..100]     | Validate [101..120]
Fold 2: Train [1..120]     | Validate [121..140]
Fold 3: Train [1..140]     | Validate [141..160]
```

The training window grows with each fold (expanding window). An alternative is a *sliding window*, where the training window has fixed size and moves forward — appropriate when the data-generating process is non-stationary and older data is less relevant.

```
from sklearn.model_selection import TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5, gap=0)   # gap prevents leakage between folds
for train_idx, val_idx in tscv.split(X):
    X_tr, X_val = X[train_idx], X[val_idx]
    y_tr, y_val = y[train_idx], y[val_idx]
    # fit and evaluate here
```

6.5.5. Normalization in Time Series

Unlike tabular data, where normalization statistics are computed once over the training set, time series normalization must account for the fact that the distribution may change over time (concept drift).

Global normalization computes statistics over the entire training series. It is simple but assumes the series is stationary. For non-stationary series, early and late training data have different distributions, and a single mean/variance is not representative of either.

Instance normalization (RevIN — Reversible Instance Normalization) normalizes each input sequence independently using its own mean and variance, then denormalizes the output. This allows the model to handle arbitrary levels and scales without being confused by the absolute magnitude of the series.

```python
class RevIN(nn.Module):
    def __init__(self, num_features, eps=1e-5):
        super().__init__()
        self.eps = eps
        self.affine_weight = nn.Parameter(torch.ones(num_features))
        self.affine_bias   = nn.Parameter(torch.zeros(num_features))

    def forward(self, x, mode="norm"):
        if mode == "norm":
            self.mean = x.mean(dim=1, keepdim=True).detach()
            self.std  = x.std(dim=1, keepdim=True).detach() + self.eps
            x = (x - self.mean) / self.std
            return x * self.affine_weight + self.affine_bias
        elif mode == "denorm":
            x = (x - self.affine_bias) / self.affine_weight
            return x * self.std + self.mean
```

6.6. Production Concerns

6.6.1. Training-Serving Skew

The most important production pipeline concern is ensuring that the transformation applied to a feature during serving is *identical* to the transformation applied during training. Any discrepancy — different normalization statistics, different handling of missing values, different encoding of categories — produces a model that sees out-of-distribution inputs at serving time and degrades silently.

A **feature store** is the architectural solution: a system that computes and stores feature values once, serving the identical computation to both training pipelines (historical features) and serving systems (online, low-latency features). The feature definition is written once and executed in both contexts, eliminating the possibility of skew by construction.

6.6.2. Distribution Shift Detection

A deployed model's performance degrades when the input distribution changes — when the world changes in ways the training data did not capture. Two types of shift require different responses:

Covariate shift: $P(X)$ changes but $P(Y \mid X)$ is unchanged. The model is still correct for inputs it receives, but it is receiving inputs unlike those it was trained on. Importance weighting can correct for this without retraining.

Concept drift: $P(Y \mid X)$ changes — the underlying relationship between features and labels has shifted. The model is genuinely wrong on the new distribution. Retraining is required.

Detection uses statistical tests on feature distributions:

```python
from scipy.stats import ks_2samp

def detect_drift(reference_df, current_df, threshold=0.05):
    drifted = {}
    for col in reference_df.columns:
        stat, p = ks_2samp(reference_df[col].dropna(),
                           current_df[col].dropna())
        if p < threshold:
            drifted[col] = {"ks_statistic": round(stat, 4),
                            "p_value": round(p, 4)}
    return drifted
```

The **Population Stability Index (PSI)** measures the magnitude of distributional shift:

$$\text{PSI} = \sum_{b} \left(P_{\text{actual},b} - P_{\text{reference},b}\right) \log \frac{P_{\text{actual},b}}{P_{\text{reference},b}}$$

PSI < 0.1 indicates negligible shift; 0.1–0.25 moderate shift requiring investigation; above 0.25 requires immediate model review.

6.7. Pipeline Comparison by Modality

Concern	Tabular	NLP	Vision	Time Series
Core representation	Scaled numerical + encoded categorical	Token IDs + attention masks	Normalized pixel tensors	Lag features + rolling statistics
Normalization	StandardScaler / RobustScaler	Per-corpus vocabulary statistics	ImageNet mean/std	Global or per-instance (RevIN)
Augmentation	SMOTE, Gaussian noise, Mixup	Synonym replacement, back-translation	Geometric + photometric transforms, MixUp, CutMix	Window jitter, time warping

Concern	Tabular	NLP	Vision	Time Series
Splitting	Stratified k-fold or group k-fold	Usually random; group split for user-level tasks	Stratified k-fold	Walk-forward (expanding or sliding window)
Leakage risk	Target encoding, feature timestamp	Label contamination in pretraining data	Test images in training split	Any feature using values after the prediction timestamp
Pretrained representations	Rare (embeddings for categorical)	Always (BERT, GPT family)	Almost always (ImageNet backbone)	Emerging (time series foundation models)

Interview Focus

Design a pipeline from scratch: given a new dataset and task, walk through every stage — what transformations apply, why, and in what order. **Leakage diagnosis:** given a pipeline that shows unexpectedly high offline metrics, identify three places where leakage could be hiding. **Modality differences:** explain why you cannot use standard k-fold CV for time series data, and describe the correct alternative. **Normalization choice:** when would you prefer RobustScaler over StandardScaler for a tabular feature, and why? **Transfer learning strategy:** describe the fine-tuning protocol for a small vision dataset (500 examples per class) starting from an ImageNet-pretrained ResNet-50.

Part III.

Module 3 - Classical Machine Learning

7. Machine Learning Models

The goal of this lecture is to implement Box–Jenkins methodology for nonstationary time series, by applying differences and fitting an ARMA model to the stationary differenced series. You should be able to recognize when the differencing is needed based on the time series and ACF plots.

Objectives

1. List the steps of Box–Jenkins methodology.
2. Identify the order of differences d and order of seasonal differences D from the plots of original and differenced time series.
3. Recall how the orders p and q are identified using ACF and PACF plots for non-seasonal time series.
4. Identify the orders P and Q for seasonal time series.

Reading materials

- Chapters 6.1–6.5 in Brockwell and Davis (2002)

7.1. Introduction to ARIMA

We have already discussed the class of ARMA models for representing stationary series. A generalization of this class, which incorporates a wide range of nonstationary series, is provided by the *autoregressive integrated moving average* (ARIMA) processes, i.e., processes which, after differencing finitely many times, reduce to ARMA processes.

If d is a nonnegative integer, then X_t is an ARIMA(p, d, q) process if $Y_t = (1 - B)^d X_t$ is a causal ARMA(p, q) process. The process is stationary if $d = 0$, in which case it reduces to an ARMA(p, q) process.

For example X_t is an ARIMA(1,1,0) process, then Y_t representing the series of its first-order differences (because $d = 1$) is an ARMA(1,0) process

$$Y_t = (1 - B)X_t = \phi_1 Y_{t-1} + Z_t,$$

where $|\phi_1| < 1$ and Z_t is white noise.

An equation for ARIMA(p, d, q) is

$$(1 - B)^d (1 - \phi_1 B - \cdots - \phi_p B^p) X_t = (1 + \theta_1 B + \cdots + \theta_q B^q) Z_t, \tag{7.1}$$

where B is the backshift operator, d is the order of differences, $\phi_1, \ldots, \phi_p$ are autoregression coefficients, p is the autoregressive order, $\theta_1, \ldots, \theta_q$ are moving average coefficients, q is the moving average order, and $Z_t \sim \text{WN}(0, \sigma^2)$. The left part of Equation 7.1 consists of the differences and the AR part; the right part represents the MA part.

> **Note**
>
> Modifications of Equation 7.1 exists, such as those having both the AR and MA parts on the right side. This affects the signs of estimated coefficients $\phi_1, \dots, \phi_p$ and possibly $\theta_1, \dots, \theta_q$. Check the help files of the software to know the exact form of the model it is estimating.

Why the process is called 'integrated'?

Recall the geometric interpretation of the integral of a curve $y = f(x)$ defined on continuous x. The integral of y corresponds to the area under the curve. For example, if $y = f(x)$ is a function describing income y based on working time x, the integral corresponds to the total income in a certain period.

In our lectures, we deal with time series defined using discrete times t (for example, years), so yearly income is Y_t, and the total income over several years can be obtained by summing the individual annual incomes, $\sum Y_t$. Hence, here we integrate by summation.

Recall the definition of random walk series, which is a cumulative sum of i.i.d. noise:

$$X_t = \sum_{i=1}^{t} Y_i,$$

where $Y_t \sim \text{i.i.d.}(0, \sigma^2)$. This X_t is the simplest example of integrated series. The notation $X_t \sim \text{I}(1)$ means X_t is a first-order integrated series. The differences of X_t

$$\begin{aligned}(1 - B)X_t &= X_t - BX_t && (7.2)\\ &= X_t - X_{t-1} && (7.3)\\ &= Y_t && (7.4)\end{aligned}$$

give us back the uncorrelated series Y_t, hence the process X_t is an ARIMA(0,1,0) process.

> **Note**
>
> ARIMA(p, d, q) processes with $d \geqslant 1$ are also called *difference-stationary* processes or processes with a *stochastic trend*. I.e., difference-stationary means the process is not stationary but can be made stationary by proper differencing.

7.2. Box–Jenkins methodology

The approach developed by Box and Jenkins (1976) applies the differencing to the original time series repeatedly, until a stationary time series is obtained. We will first learn how to identify the number of differences (i.e., the order of differences d) by analyzing plots of the time series and ACF at each iteration of the method. In the later lecture on detecting trends in time series, we will also introduce formal tests for the integration order.

Below is a general algorithm used to fit ARIMA(p, d, q) models

1. Start assuming $d = 0$.
2. Plot the time series and ACF.

3. If the plots suggest nonstationarity, iterate differencing and plotting to update d:
    - Apply differences of the current time series, write $d = d + 1$
    - Plot the time series and ACF
    - If nonstationarity is still obvious, repeat the differencing and plotting
4. Identify the orders p and q from the plots of ACF and PACF of the latest (differenced) time series.
5. Estimate the model.
6. Apply model diagnostics, particularly for homogeneity, uncorrelatedness, and normality of residuals. Address the violations by respecifying the model.
7. Forecast with the resultant model.

Using the Box–Jenkins methodology, the linear predictor approximately follows a normal distribution, i.e.,

$$\hat{X}_{n+h} \sim N\left(X_{n+h}, \operatorname{var}(\hat{X}_{n+h})\right).$$

Therefore, a $(100 - \alpha)\%$ prediction interval is

$$\hat{X}_{n+h} \pm z_{1-\alpha/2}\sqrt{\operatorname{var}(\hat{X}_{n+h})}.$$

Example: ARIMA for Lake Baikal

Here we find an ARIMA model for the Lake Baikal thaw (breakup) dates from the Global Lake and River Ice Phenology Database (Benson et al. 2020).

```
# Calculate calendar day from ice break-up date
B <- read.csv("data/baikal.csv", skip = 1) %>%
    mutate(Date_iceoff = as.Date(paste(iceoff_year, iceoff_month, iceoff_day,
                                       sep = "-"))) %>%
    mutate(DoY_iceoff = as.numeric(format(Date_iceoff, "%j")))

# Convert to ts format
iceoff <- ts(B$DoY_iceoff, start = B$iceoff_year[1])
```

Figure 7.1 shows that there is possibly a decreasing trend (ice melts earlier in the year), although the ACF declines fast.

```
X <- iceoff
p1 <- forecast::autoplot(X) +
    xlab("Year") +
    ylab("Ice breakup day")
p2 <- forecast::ggAcf(X) +
    ggtitle("")
p3 <- forecast::ggAcf(X, type = "partial") +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

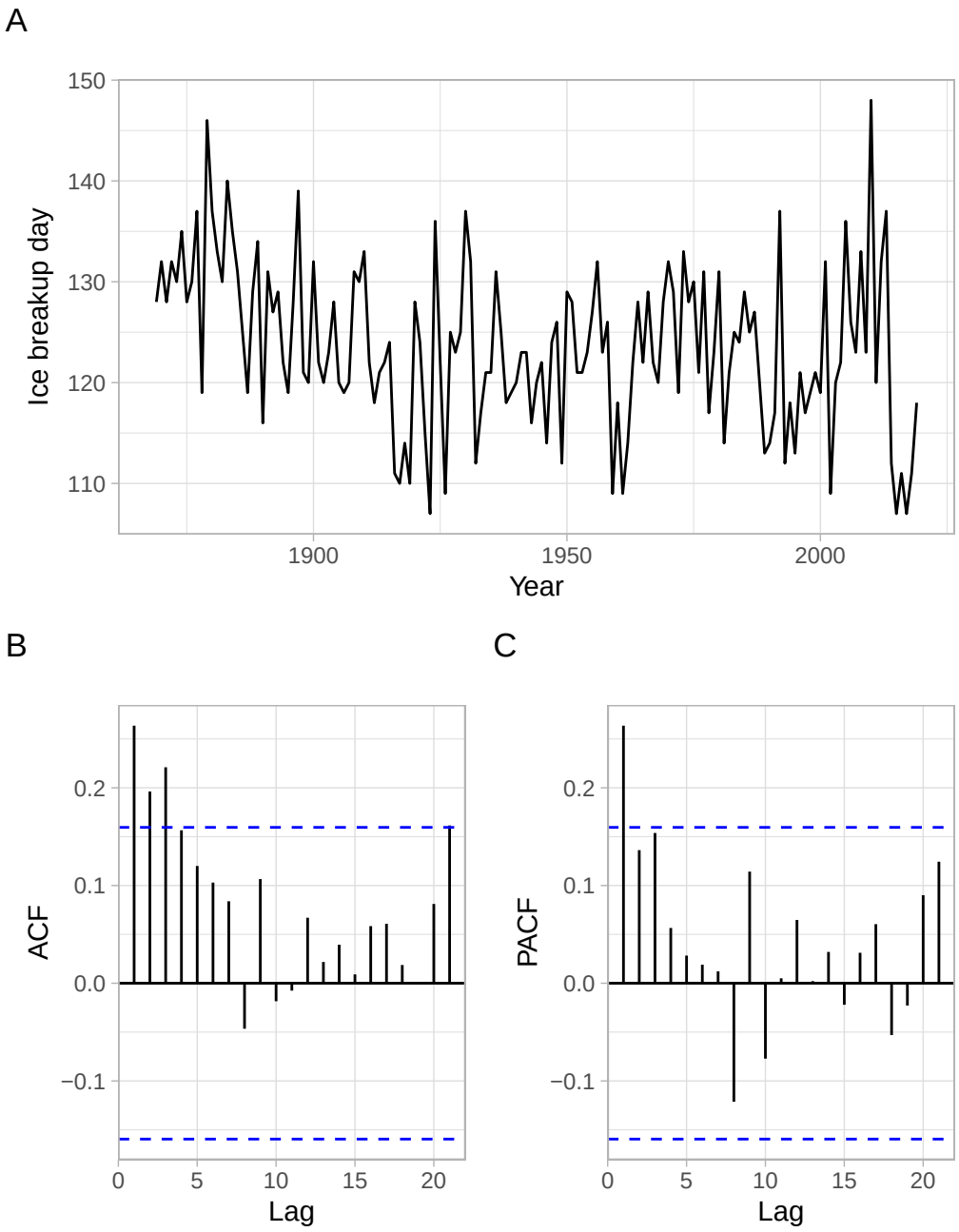


Figure 7.1.: Plots for the original time series of ice breakup days.

To remove this potential trend, we apply differencing once (more specifically, consecutive differencing, $X_t - X_{t-1}$) and produce the plots again. From Figure 7.2, there is no tendency in the differenced series, and the ACF declines fast, so we achieved stationarity and no need to difference the data more. Overall, we differenced the time series once to achieve stationarity, so the order of differences $d = 1$.

```
X <- diff(iceoff)
p1 <- ggplot2::autoplot(X) +
    xlab("Year") +
    ylab("diff(Ice breakup day)")
p2 <- forecast::ggAcf(X) +
    ggtitle("")
p3 <- forecast::ggAcf(X, type = "partial") +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

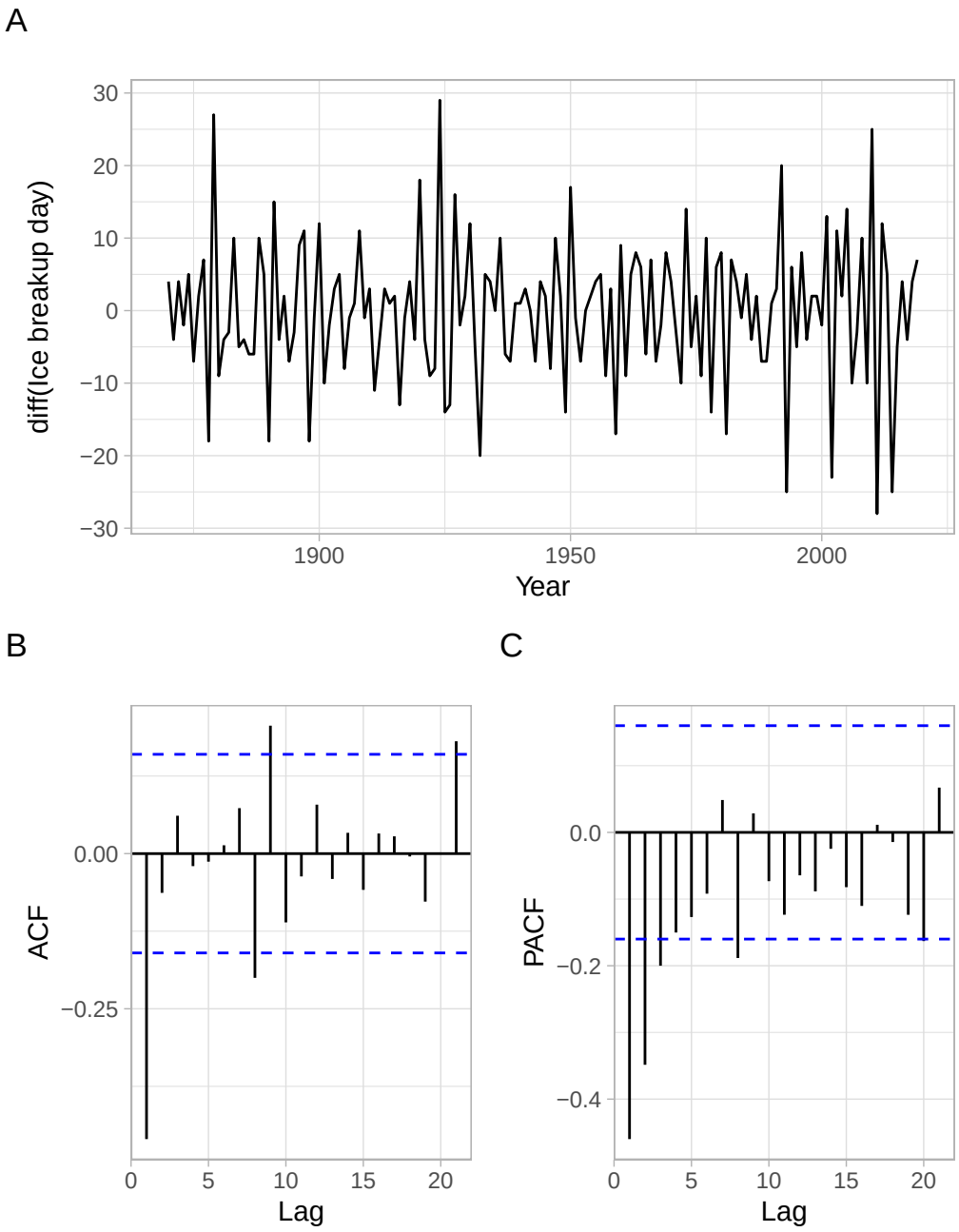


Figure 7.2.: Plots for the differenced time series of ice breakup days.

Continue working with Figure 7.2 to identify the orders p and q. If we treat the behavior of ACF as exhibiting a cut-off, and PACF having an exponential decay, an MA(q) model might be plausible (i.e., $p = 0$). Since the ACF cuts off after lag 1, $q = 1$ in this case.
Hence, we specified the model for ice breakup dates to be ARIMA(0,1,1), which can be written as

$$(1 - B)Y_t = (1 + \theta_1 B)Z_t$$

or

$$Y_t = Y_{t-1} + \theta_1 Z_{t-1} + Z_t,$$

where Y_t represents the ice breakup dates in the year t, θ_1 is the moving average coefficient, and $Z_t \sim \text{WN}(0, \sigma^2)$.
We can now estimate the model, for example, using `stats::arima()`.

```
mod_baikal <- stats::arima(iceoff, order = c(0, 1, 1))
mod_baikal
```

```
#>
#> Call:
#> stats::arima(x = iceoff, order = c(0, 1, 1))
#>
#> Coefficients:
#>          ma1
#>       -0.843
#> s.e.   0.075
#>
#> sigma^2 estimated as 59:  log likelihood = -519,  aic = 1042
```

We can also check the orders selected automatically:

```
forecast::auto.arima(iceoff)
```

```
#> Series: iceoff
#> ARIMA(0,1,1)
#>
#> Coefficients:
#>          ma1
#>       -0.843
#> s.e.   0.075
#>
#> sigma^2 = 59.4:  log likelihood = -519
#> AIC=1042   AICc=1043   BIC=1048
```

In the next step, we apply diagnostic checks for the residuals, for example, using plots (Figure 7.3). Remember that the residuals should resemble white noise.

```
e <- mod_baikal$residuals
p1 <- ggplot2::autoplot(e) +
    ylab("Residuals")
p2 <- forecast::ggAcf(e) +
    ggtitle("")
p3 <- ggpubr::ggqqplot(e) +
    xlab("Standard normal quantiles")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

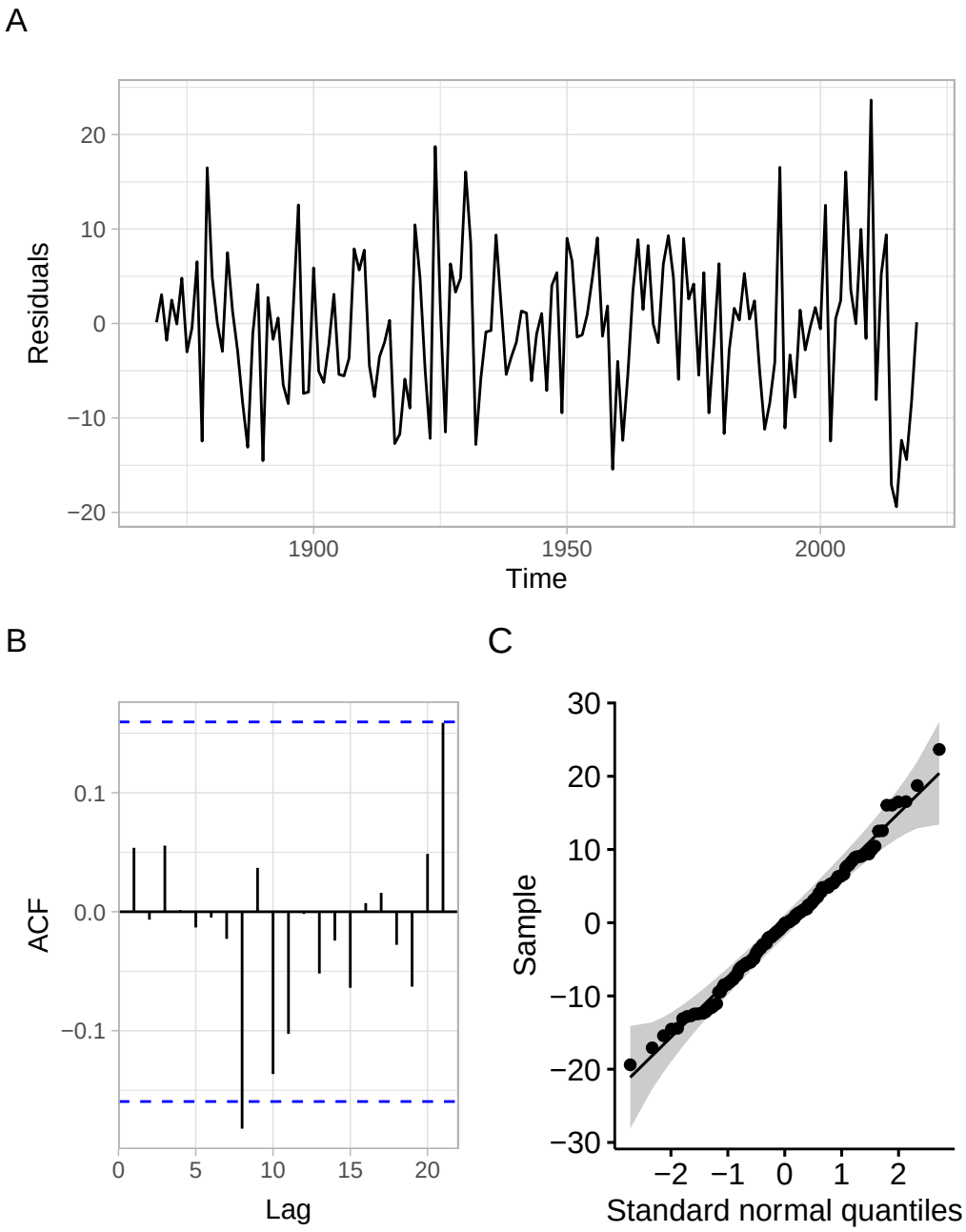


Figure 7.3.: Residual diagnostics for the ARIMA(0,1,1) model for ice breakup dates.

Given that the diagnostics plots show satisfactory behavior of the residuals (Figure 7.3), continue with forecasting using this model (Figure 7.4). Note that ARIMA(0,1,1) is mathematically equivalent to simple exponential smoothing, hence the forecast is a horizontal line.

```
ggplot2::autoplot(forecast::forecast(mod_baikal, h = 10)) +
    xlab("Year") +
    ylab("Ice breakup day") +
    ggtitle("")
```

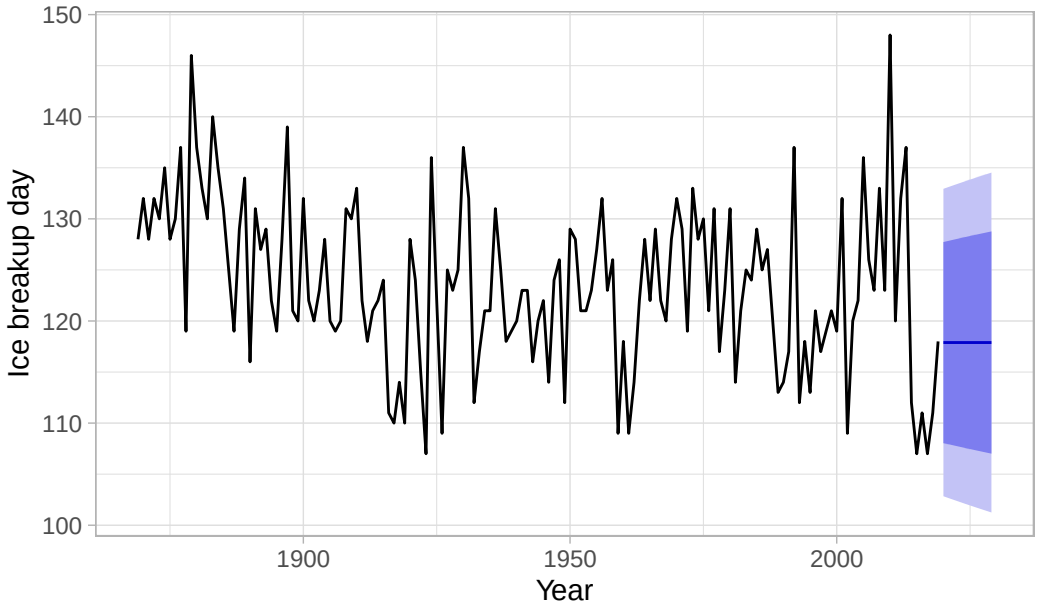


Figure 7.4.: ARIMA(0,1,1) model predictions of ice breakup dates 10 years ahead.

7.3. Seasonal ARIMA (SARIMA)

Recall that time series with seasonal variability or another strictly periodic component (e.g., daily cycles) can be deseasonalized by applying differencing that is not of consecutive values but with a lag equal to the period of the cyclical variability. We will use such differences to remove *strong* periodicity, as an additional step in the Box–Jenkins algorithm.

Similar to the regular differences, we will apply seasonal differences not to eliminate autocorrelations at the corresponding lags, but to achieve fast decay of ACF at seasonal lags. Even after the seasonal differences, there might be significant spikes at the seasonal lags in ACF and PACF, which can be addressed by selecting proper seasonal autoregressive and moving average orders. Therefore, we can define orders of integration, AR, and MA for the seasonal part of the time series in the same way we define these orders for the non-seasonal part.

Based on the definitions in Brockwell and Davis (2002), X_t is a *seasonal autoregressive integrated moving average*, SARIMA$(p,d,q)(P,D,Q)_s$ process if the differenced series $Y_t = (1-B)^d(1-B^s)^D X_t$ is a causal ARMA process. Here d and D are nonnegative integers, and s is the period.

> **Note**
>
> In practice, $D \leqslant 1$ and $P, Q \leqslant 3$.

An equation for SARIMA$(p, d, q)(P, D, Q)_s$ is

$$\begin{aligned}(1-B)^d(1-\phi_1 B - \cdots - \phi_p B^p)(1-B^s)^D(1-\Phi_1 B^s - \cdots - \Phi_P B^{sP})X_t \\ = (1+\theta_1 B + \cdots + \theta_q B^q)(1+\Theta_1 B^s + \cdots + \Theta_Q B^{sQ})Z_t,\end{aligned} \tag{7.5}$$

where D is the order of seasonal differences, $\Phi_1, \ldots, \Phi_P$ are the seasonal autoregression coefficients, P is the seasonal autoregressive order, $\Theta_1, \ldots, \Theta_q$ are seasonal moving average coefficients, Q is the seasonal moving average order, and the rest of the terms are the same as in Equation 7.1.

> Example: SARIMA for the number of airline passengers
>
> Here we revisit the time series on airline passengers from which we have removed the trend by taking regular non-seasonal differences once ($d = 1$). Notice from Figure 7.5 C and D how after taking the usual differences the upward trend disappeared, and ACF started to decay much faster. At the seasonal lags, however, the ACF decays still linearly (Figure 7.5 D), which suggested differencing at the seasonal lag to remove strong periodicity. After taking seasonal differences once ($D = 1$), the time series looks stationary (Figure 7.5 E), and the ACF decays fast at both seasonal and non-seasonal lags. This is enough differencing.

```r
Yt <- log10(AirPassengers)

# Apply first-order (non-seasonal) differences
D1 <- diff(Yt)

# Additionally, apply first-order seasonal differences
D1D12 <- diff(D1, lag = 12)

p1 <- ggplot2::autoplot(Yt) +
    xlab("Year") +
    ylab("lg(Air passangers)") +
    ggtitle("Yt")
p2 <- forecast::ggAcf(Yt) +
    ggtitle("Yt") +
    xlab("Lag (months)")
p3 <- ggplot2::autoplot(D1) +
    xlab("Year") +
    ylab("lg(Air passangers)") +
    ggtitle("(1-B)Yt")
p4 <- forecast::ggAcf(D1) +
    ggtitle("(1-B)Yt") +
    xlab("Lag (months)")
p5 <- ggplot2::autoplot(D1D12) +
    xlab("Year") +
    ylab("lg(Air passangers)") +
    ggtitle("(1-B)(1-B12)Yt")
p6 <- forecast::ggAcf(D1D12) +
    ggtitle("(1-B)(1-B12)Yt") +
    xlab("Lag (months)")
(p1 + p2) / (p3 + p4) / (p5 + p6) +
    plot_annotation(tag_levels = 'A')
```

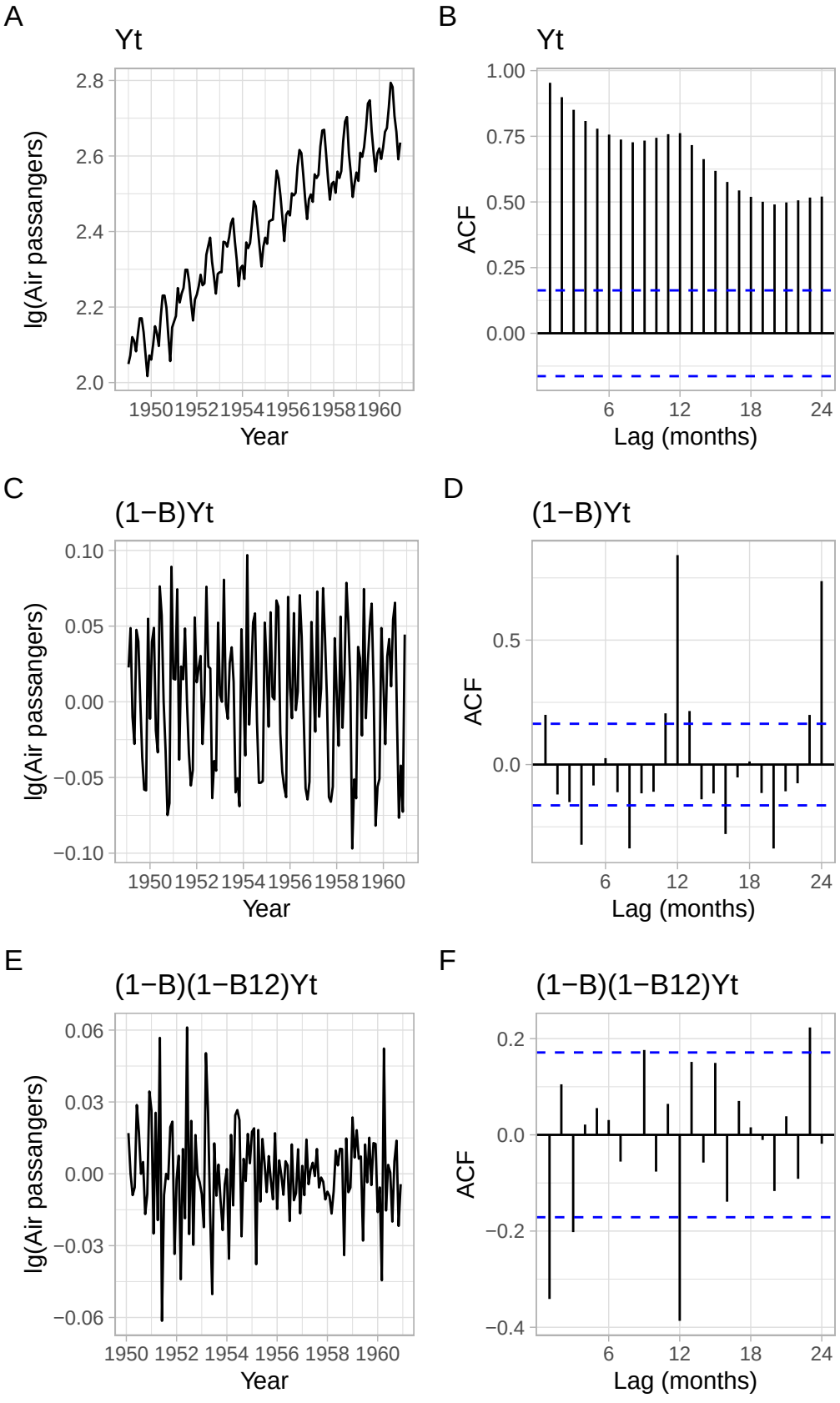


Figure 7.5.: Time series plot of the airline passenger series with an estimated ACF and the detrended (differenced) series with their ACFs.

For the next step, see Figure 7.6 to identify the orders p and q. For that, look only at the non-seasonal lags, 1–11. Both ACF and PACF have significant values at lags 1 and 3, which could correspond to AR(3), MA(3), ARMA(1,1), or ARMA with higher orders. From these options ARMA(1,1) has fewer parameters, hence we prefer this model, with $p = 1$ and $q = 1$, as the most parsimonious option. (However, given that the correlations at lag 2 are not statistically significant, information criteria may penalize adding extra arguments and might prefer a more compact specification, AR(1) or MA(1).)
Next, use Figure 7.6 again to identify orders P and Q. Now look only at the seasonal lags 12, 24, 36, etc. Both the ACF and PACF have significant values only on the first of those lags (lag 12), which could correspond to AR(1), MA(1), or ARMA(1,1) for the seasonal component. From these options, AR(1) and MA(1) are the most parsimonious so we should select one of them or use some numeric criterion to select the best model (e.g., information criterion like AIC or forecasting accuracy on a testing set).

```
p6 <- forecast::ggAcf(D1D12, lag.max = 36) +
    ggtitle("(1-B)(1-B12)Yt") +
    xlab("Lag (months)")
p7 <- forecast::ggAcf(D1D12, lag.max = 36, type = "partial") +
    ggtitle("(1-B)(1-B12)Yt") +
    xlab("Lag (months)")
p6 + p7 +
    plot_annotation(tag_levels = 'A')
```

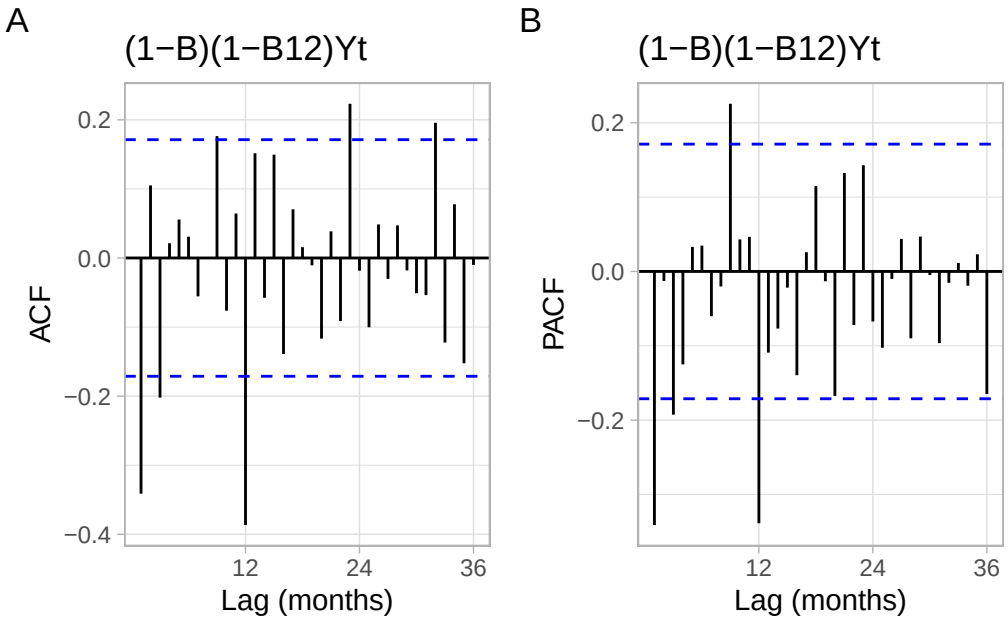


Figure 7.6.: ACF and PACF of the differenced airline passenger time series $(1-B)(1-B^{12})Y_t$.

Overall, our analysis suggested that SARIMA(1,1,1)(1,1,0) or SARIMA(1,1,1)(0,1,1) is a plausible model for this time series. For example, estimate SARIMA(1,1,1)(0,1,1):

```
mod_air <- stats::arima(Yt, order = c(1, 1, 1),
                        seasonal = list(order = c(0, 1, 1),
                                        period = 12))
mod_air
```

```
#>
#> Call:
#> stats::arima(x = Yt, order = c(1, 1, 1), seasonal = list(order = c(0, 1, 1),
#>     period = 12))
#>
#> Coefficients:
#>          ar1     ma1    sma1
#>       0.196  -0.578  -0.564
#> s.e.  0.247   0.213   0.075
#>
#> sigma^2 estimated as 0.000253:  log likelihood = 354,  aic = -700
```

SARIMA$(1,1,1)(0,1,1)_{12}$ model can be written down as

$$\begin{aligned}(1-B)(1-\phi_1 B)(1-B^{12})X_t \\ = (1+\theta_1 B)(1+\Theta_1 B^{12})Z_t.\end{aligned}$$

We can apply the backshift operations are rewrite the model without the backshift operator as follows:

$$\begin{aligned}(1-B-\phi_1 B+\phi_1 B^2)(1-B^{12})X_t \\ = (1+\theta_1 B+\Theta_1 B^{12}+\theta_1\Theta_1 B^{13})Z_t,\end{aligned}$$

then

$$\begin{aligned}(1-B-\phi_1 B+\phi_1 B^2-B^{12}+B^{13}+\phi_1 B^{13}-\phi_1 B^{14})X_t \\ = (1+\theta_1 B+\Theta_1 B^{12}+\theta_1\Theta_1 B^{13})Z_t,\end{aligned}$$

then

$$\begin{aligned}X_t - X_{t-1} - \phi_1 X_{t-1} + \phi_1 X_{t-2} - X_{t-12} + X_{t-13} + \phi_1 X_{t-13} - \phi_1 X_{t-14} \\ = Z_t + \theta_1 Z_{t-1} + \Theta_1 Z_{t-12} + \theta_1\Theta_1 Z_{t-13},\end{aligned}$$

then

$$\begin{aligned}X_t = X_{t-1} + \phi_1 X_{t-1} - \phi_1 X_{t-2} + X_{t-12} - X_{t-13} - \phi_1 X_{t-13} + \phi_1 X_{t-14} \\ +\theta_1 Z_{t-1} + \Theta_1 Z_{t-12} + \theta_1\Theta_1 Z_{t-13} + Z_t,\end{aligned}$$

where $X_t = \lg(AirPassengers)$ and $Z_t \sim \mathrm{WN}(0,\sigma^2)$.
Below are results based on the automatic selection of the orders:

```
forecast::auto.arima(Yt)
```

```
#> Series: Yt
#> ARIMA(0,1,1)(0,1,1)[12]
#>
#> Coefficients:
#>          ma1    sma1
#>       -0.402  -0.557
#> s.e.   0.090   0.073
#>
#> sigma^2 = 0.000259:  log likelihood = 354
#> AIC=-702   AICc=-702   BIC=-693
```

The orders selected automatically based on AIC suggest that indeed the non-significance of correlations at lag 2 and relatively low correlations at lag 3 made it not worthy to estimate additional parameters in the non-seasonal part, for which MA(1) specification was selected, not the suggested ARMA(1,1), AR(3), or MA(3).
In the next step, we apply diagnostic checks for the residuals, for example, using plots (Figure 7.7).

```
e <- mod_air$residuals
p1 <- ggplot2::autoplot(e) +
    ylab("Residuals")
p2 <- forecast::ggAcf(e) +
    ggtitle("")
p3 <- ggpubr::ggqqplot(e) +
    xlab("Standard normal quantiles")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

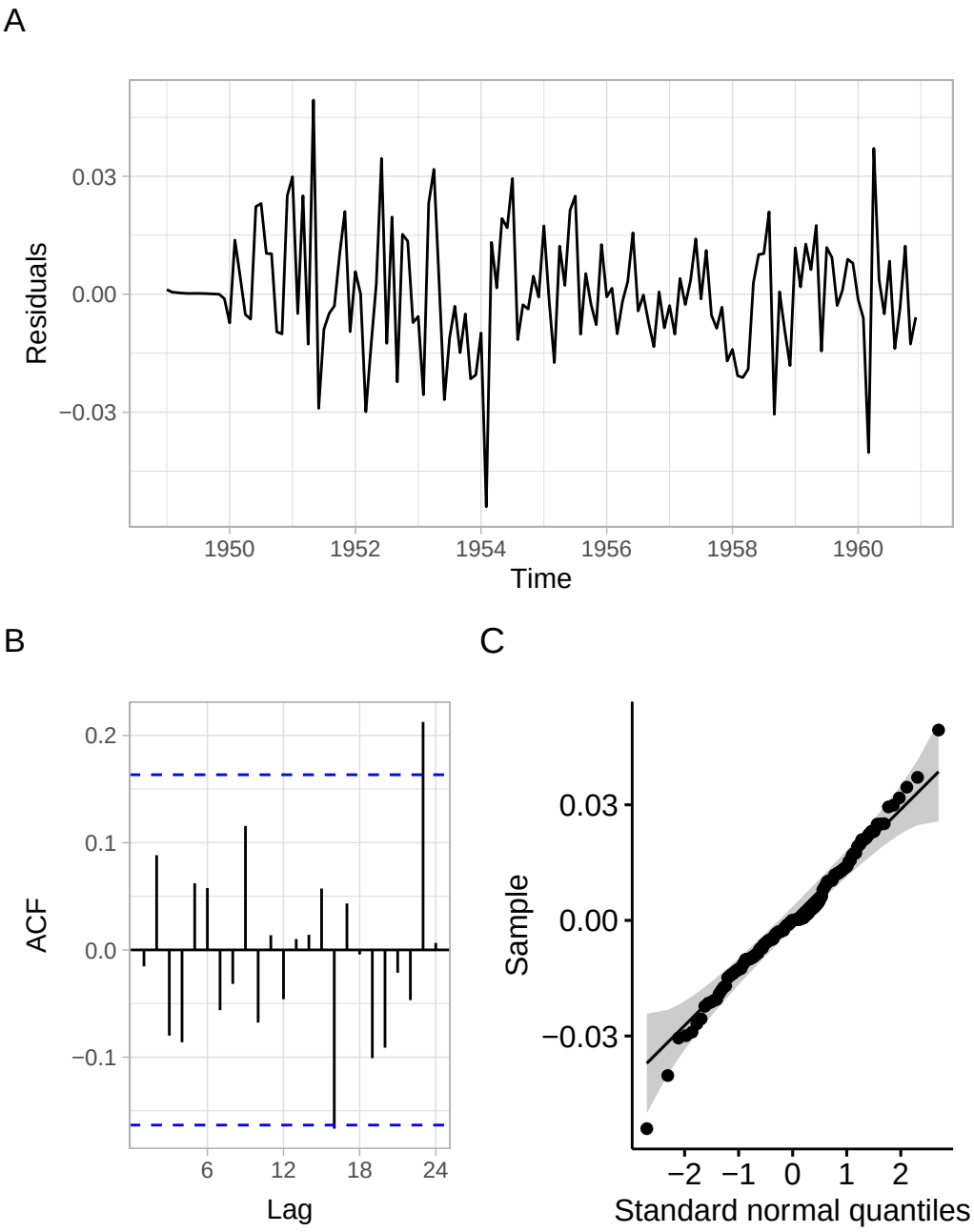


Figure 7.7.: Residual diagnostics for the SARIMA model for the airline passenger data.

Given that the diagnostics plots show satisfactory behavior of the residuals (Figure 7.7), continue with forecasting using this model (Figure 7.8).

```
ggplot2::autoplot(forecast::forecast(mod_air, h = 24)) +
    xlab("Year") +
    ylab("lg(Air passangers)") +
    ggtitle("")
```

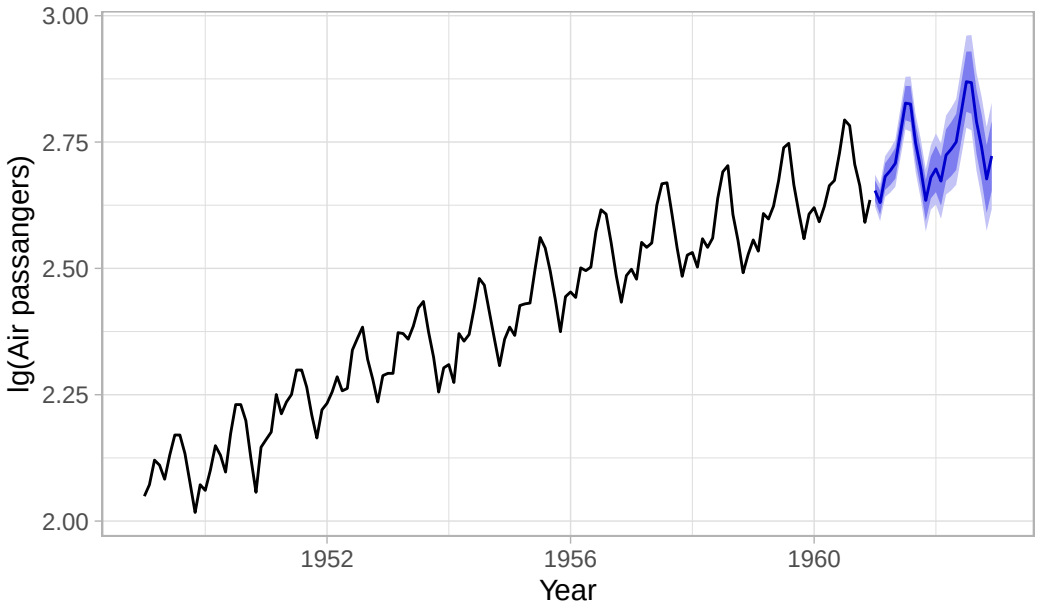


Figure 7.8.: SARIMA model predictions of airline passenger data 2 years ahead.

Compare ACF in Figure 7.7 with ACF of regression residuals in **?@fig-airpassangersRegTrendSeas** and **?@fig-airpassangersRegTrendFourier**. The residuals of those regression models are autocorrelated, while the models have more parameters than the specified SARIMA models. Hence, the SARIMA model is much better for this time series.

7.4. Conclusion

In this lecture, we discovered ARIMA as an extension of ARMA modeling to nonstationary data and SARIMA as an extension of ARIMA models to time series with periodicity (seasonality).

We learned Box–Jenkins iterative procedure for identifying such models, including the orders of differences, d and D, and p, q, P, and Q.

Please remember to specify the criterion for selecting a model beforehand and consider such options as cross-validation and using a testing set.

7.5. Appendix

Equivalences

Mathematically, some models are equivalent one to another. Below are some examples.

- Forecasts of ARIMA(0,1,1) are equivalent to simple exponential smoothing.

An ARIMA(0,1,1) model can be written as

$$Y_t = Y_{t-1} + \theta_1 Z_{t-1} + Z_t,$$

from which the one-step-ahead forecast is

$$\hat{Y}_t = Y_{t-1} + \theta_1 (Y_{t-1} - \hat{Y}_{t-1}).$$

If we define $\theta_1 = \alpha - 1$, then the above equation transforms to

$$\begin{aligned} \hat{Y}_t &= Y_{t-1} + (\alpha - 1)(Y_{t-1} - \hat{Y}_{t-1}) \\ &= Y_{t-1} + (\alpha - 1)Y_{t-1} - (\alpha - 1)\hat{Y}_{t-1} \\ &= \alpha Y_{t-1} + (1 - \alpha)\hat{Y}_{t-1} \end{aligned}$$

- Forecasts of ARIMA(0,2,2) are equivalent to Holt's method.
- Forecasts of SARIMA$(0,1,s+1)(0,1,0)_s$ are equivalent to Holt–Winters additive method.

8. Time-Series Analysis and Models

This lecture demonstrates the effects of autocorrelation on the results of statistical tests for trend detection. You will recall the assumptions of the classical t-test and Mann–Kendall tests and will be able to suggest bootstrapped modifications of these tests to overcome the problem of temporal dependence. Moreover, you will become familiar with tests for non-monotonic parametric trends and stochastic trends.

Objectives

1. Recall the form and standard assumptions of the classical t-test and Mann–Kendall tests.
2. Learn about bootstrapping for time series, particularly the approaches that preserve temporal dependency in the data (e.g., sieve and block bootstraps).
3. Apply bootstrap to the t-test and Mann–Kendall tests.
4. Test for non-monotonic parametric trends.
5. Test for stochastic trends (i.e., unit roots).

Reading materials

- Chapter 8.1 in Chatterjee and Hadi (2006) and Chapters 5.1–5.2 in Chatterjee and Simonoff (2013) about the effects of autocorrelation
- Bühlmann (2002) on bootstrap for time series
- Chapter 6.3 in Brockwell and Davis (2002) on unit-root tests

8.1. Introduction

The majority of studies focus on the detection of linear or monotonic trends, using classical t-test or rank-based Mann–Kendall test, typically under the assumption of uncorrelated data.

There exist two main problems:

1. dependence effect, i.e., the issue of inflating significance due to dependent observations – a possible remedy is to employ bootstrap (Noguchi et al. 2011; Cabilio et al. 2013);
2. changepoints or regime shifts that affect the linear or monotonic trend hypothesis (Seidel and Lanzante 2004; Powell and Xu 2011; Lyubchich 2016).

Hence, our goal is to provide reliable inference even for dependent observations and to test different alternative trend shapes.

8.2. 'Traditional' tests assuming independence

8.2.1. Student's t-test for linear trend

The Student's t-test for linear trend uses the regression model of linear trend

$$Y_t = b_0 + b_1 t + e_t,$$

where b_0 and b_1 are the regression coefficients, t is time, and e_t are regression errors typically assumed to be homoskedastic, uncorrelated, and normally distributed.

The test hypotheses are

H_0: no trend ($b_1 = 0$)
H_1: linear trend ($b_1 \neq 0$)

Figure 8.1 shows a simulated stationary time series $Y_t \sim$ AR(1) of length 100 (notice the 'burn-in' period in simulations).

```
set.seed(123)
Y <- arima.sim(list(order = c(1,0,0), ar = 0.5), n = 100, n.start = 100)
forecast::autoplot(Y)
```

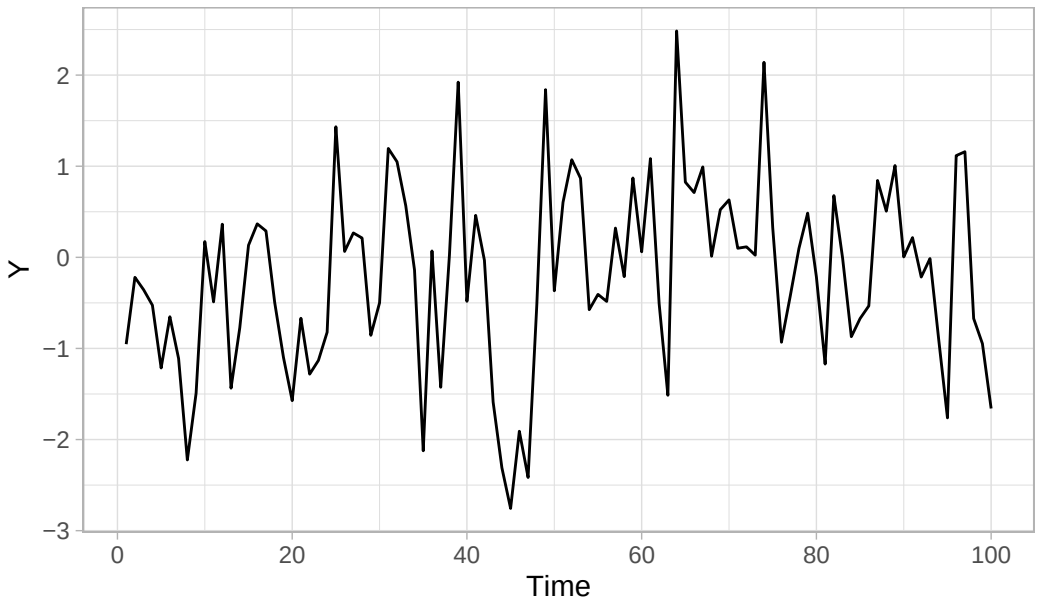


Figure 8.1.: A simulated stationary AR(1) series of length 100.

Apply the t-test to this time series Y_t:

```r
t <- 1:length(Y)
mod <- lm(Y ~ t)
summary(mod)
```

```
#>
#> Call:
#> lm(formula = Y ~ t)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.5115 -0.6106  0.0166  0.6808  2.5868
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.57984    0.20182   -2.87    0.005 **
#> t            0.00746    0.00347    2.15    0.034 *
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 1 on 98 degrees of freedom
#> Multiple R-squared:  0.045,  Adjusted R-squared:  0.0352
#> F-statistic: 4.62 on 1 and 98 DF,  p-value: 0.0341
```

Type I error (false positives) of this test is inflated due to the dependence effect (the assumption of uncorrelatedness is violated). Additionally, this test is limited only to detecting linear trends (see the alternative hypothesis H_1).

8.2.2. Mann–Kendall test for monotonic trend

The Mann–Kendall test is based on the Kendall rank correlation and is used to determine if a non-seasonal time series has a monotonic trend over time.

H_0: no trend
H_1: monotonic trend

Test statistic:

$$S = \sum_{k=1}^{n-1} \sum_{j=k+1}^{n} sgn(X_j - X_k),$$

where $sgn(x)$ takes on the values of 1, 0, and -1, for $x > 0$, $x = 0$, and $x < 0$, respectively.

Kendall (1975) showed that S is asymptotically normally distributed and, for the situations where there may be ties in the X values,

$$\begin{aligned} \mathrm{E}(S) &= 0, \\ \mathrm{var}(S) &= \frac{1}{18}\left[n(n-1)(2n+5) - \sum_{j=1}^{p} t_j(t_j-1)(2t_j+5)\right], \end{aligned}$$

where p is the number of tied groups in the time series, and t_j is the number of data points in the jth tied group.

Its seasonal version is the sum of the statistics for individual seasons over all seasons (Hirsch et al. 1982):

$$S = \sum_{j=1}^{m} S_j.$$

For data sets as small as $n = 2$ and $m = 12$, the normal approximation of the test statistic is adequate and thus the test is easy to use. The method also accommodates both

- a moderate number of missing observations and
- values below the detection limit, as the latter are treated as ties (see details in Esterby 1996).

To apply the test, use the package `Kendall`. Also, note the statement about bootstrap in the help file for the function `Kendall::MannKendall()`.

```
Kendall::MannKendall(Y)
```

```
#> tau = 0.13, 2-sided pvalue =0.06
```

This test is still limited to only monotonic trends and independent observations.

8.3. Introduction to bootstrap

8.3.1. Bootstrap for independent data

When some of the statistical assumptions are violated, we may switch to using other methods or try to accommodate these violations by modifying part of the existing method.

- One of the assumptions of Pearson correlation analysis is the absence of outliers. When outliers are in the data, we can opt for using Spearman correlations based on ranks, not the actual values. Thus, here we *switch the method*.
- We often use the normal distribution for inference about the population mean because we assume that averages of independent samples coming from the same population are distributed normally. The normal approximation is true for large samples or for small samples if the underlying population is close to being normally distributed. So what happens if we have a small sample from a non-normal population? We do not know the distributional law under which sample means are distributed in this case, but we can *approximate the distribution using bootstrapped statistics*.

The seminal paper by Efron (1979) describes bootstrap for i.i.d. data. In two words, the idea is the following: we can relax distributional assumptions and reconstruct the distribution of the sample statistic by *resampling data with replacement* and recalculating the statistic over and over. The resampling step will give us artificial 'new' samples, while the statistics calculated multiple times on those samples will let us approximate the distribution of the statistic of interest. Thus, by repeating

the resampling and estimation steps many times, we will know how the statistic is distributed, even if the sample is small and not normally distributed.

Let x_i $(i = 1, \dots, n)$ be sample values collected from a non-normally distributed population using simple random sampling. If the sample size n is small, we cannot be sure that the distribution of sample averages $\bar{x}$ is normal, so we will approximate it using the following bootstrapping steps:

1. Let x_i^* $(i = 1, \dots, n)$ be a sample with a replacement from the original sample x_i. This is a way for us, without knowing the underlying distribution of the population and without collecting new data, to get a new artificial sample from the population.
2. Calculate the mean (or another statistic of interest) from the bootstrapped data. For example, $\bar{x}^* = n^{-1} \sum x_i^*$ is the bootstrapped mean.
3. Repeat the steps above a large number of times to obtain a distribution of the statistics $\bar{x}_1^*, \dots, \bar{x}_B^*$ (the *bootstrapped distribution*), where B is a large enough number of bootstrap replications. Typically, $B \geqslant 1000$.
4. Finally, we use the bootstrapped distribution for the intended analysis. It often involves calculating confidence intervals. There are a few ways the intervals can be calculated (see Davison and Hinkley 1997), some of the methods provide symmetric intervals, while others do not. Probably the simplest is the *percentile confidence interval*, which computes the $\alpha/2$th and $(1 - \alpha/2)$th quantiles of the bootstrapped distribution, to get an interval for confidence $1 - \alpha$.

Example: Bootstrap confidence interval for the population mean

Consider a small sample of mercury (Hg) concentrations in fish tissue (Lyubchich et al. 2016). These observations do not constitute a time series; the data were collected in such a way that we can treat the samples as independent. Figure 8.2 A shows that the underlying population of Hg concentrations is likely not normally distributed, so we will use bootstrapping to calculate a confidence interval for the mean.

```r
# Mercury concentrations
Hg <- c(10.159162, 9.190562, 7.776279, 11.417387, 8.494544,
        9.948271, 7.865391, 7.412350, 8.112304, 7.541787)

# Set seed for reproducible bootstrapping
set.seed(123)

# Number of bootstrap replications
B = 1000

# Significance level
alpha = 0.05

# Compute bootstrapped means (option 1)
xbar_star <- numeric()
for (b in 1:B) {
    # bootstrapped sample
    Hgstar <- sample(Hg, replace = TRUE)
    # bootstrapped mean
    xbar_star[b] <- mean(Hgstar)
}

# Compute bootstrapped means (option 2)
xbar_star <- sapply(1:B, function(b) mean(sample(Hg, replace = TRUE)))

# Bootstrap percentile interval for confidence 1 - alpha
interval <- quantile(xbar_star, probs = c(alpha/2, 1 - alpha/2))
interval
```

```
#>  2.5% 97.5%
#>  8.04  9.64
```

```r
p1 <- ggplot(data.frame(x = Hg), aes(x = x)) +
    geom_histogram(aes(y = after_stat(density)),
                   boundary = floor(min(Hg)), binwidth = 1,
                   fill = "grey50") +
    xlab("Hg (ng/g)") +
    ylab("Density") +
    ggtitle("Hg observations")
p2 <- ggplot(data.frame(x = xbar_star), aes(x = x)) +
    geom_histogram(aes(y = after_stat(density)), bins = 15, fill = "grey50") +
    geom_vline(xintercept = interval, lty = 2) +
    xlab("Hg (ng/g)") +
    ylab("Density") +
    ggtitle("Bootstrapped means")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

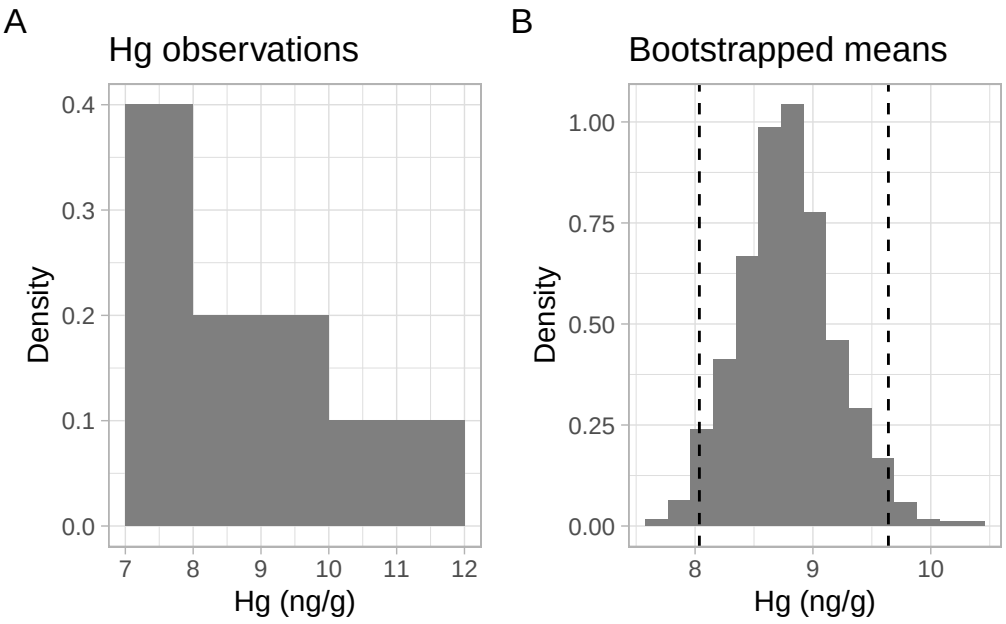


Figure 8.2.: Histograms of the original data and bootstrapped means. The dashed lines denote the bootstrap confidence interval for the mean.

The calculated confidence interval can be used in hypothesis testing. Note that the package `boot` provides functions with more options for the calculations.

We cannot apply the outlined bootstrapping directly to time series or spatial data because these data are not i.i.d. and resampling will break the order and dependence structure.

8.3.2. Bootstrap for time series

To account for the dependence structure, some modifications to the bootstrap procedure were proposed, including block bootstrap and sieve bootstrap (see Bühlmann 2002 and references therein).

While the bootstrap for independent data generates samples that mimic the underlying population distribution, the bootstrap for time series also aims to mimic or preserve the dependence structure of the time series. Hence, the first step of the outlined bootstrapping algorithm should be modified so that the generated bootstrapped samples preserve the dependence structure that we want to accommodate in our analysis. For example, if we want to accommodate serial dependence in the t-test, the bootstrapped time series should be autocorrelated similarly to the original series, so we can approximate the distribution of the test statistic when the assumption of independence is violated.

Block bootstrap works for general stationary time series or categorical series (e.g., genetic sequences). In the simplest version of the algorithm, the observed time series of length n is used to create overlapping blocks of fixed length l, which are then resampled with replacement to create a bootstrapped time series. To match the original sample size, the last block entering the bootstrapped

series can be truncated. By resampling blocks rather than individual observations, we preserve the original dependence structure within each block (Bühlmann 2002; Härdle et al. 2003).

Other versions of this algorithm include non-overlapping blocks and blocks of random length l sampled from the geometric distribution (the latter version is also known as the *stationary bootstrap*), but these versions often show poorer performance (Bühlmann 2002; Härdle et al. 2003).

The block length l should be adjusted each time to the statistic of interest, data generating process, and purpose of the estimation, such as distribution, bias, or variance estimation (Bühlmann 2002). In other words, l is the algorithm's major tuning parameter that is difficult to select automatically. Bühlmann (2002) also points out the lack of good interpretations and reliable diagnostic tools for the block length l.

Sieve bootstrap generally works for time series that are realizations of linear AR(∞) processes (Bühlmann 2002), with more recent research extending the method to all stationary purely nondeterministic processes (Kreiss et al. 2011). In this algorithm, an autoregressive (AR) model acts as a sieve by approximating the dependence structure and letting through only the i.i.d. residuals ϵ_t. We then center (subtract mean) and resample with replacement the residuals (there is no more structure we need to preserve after the sieve, so this bootstrap step is no different from resampling i.i.d. data). We introduce these bootstrapped residuals ϵ_t^* back into the AR($\hat{p}$) model to obtain a bootstrapped time series X_t^*:

$$X_t^* = \sum_{j=1}^{\hat{p}} \hat{\phi}_j X_{t-j}^* + \epsilon_t^*,$$

where $\hat{\phi}_j$ are the AR coefficients, estimated on the original time series X_t, and $\hat{p}$ is the selected AR order. The order p is the main tuning parameter in this algorithm, but as Bühlmann (2002) points out, it has several advantages compared with the block length in block bootstrap:

- The order $\hat{p}$ can be selected using the Akaike information criterion (AIC) or Bayesian information criterion (BIC) based on the available data, which means the method adapts to the sample size and underlying dependence structure.
- The AR order is more interpretable than the block length.
- There exist diagnostic procedures to check how good is the selected $\hat{p}$, for example, graphs and tests for remaining autocorrelation in the AR($\hat{p}$) residuals ϵ_t.

Other versions of sieve bootstrap include options of obtaining ϵ_t^* from a parametric distribution (e.g., normal distribution with the mean of 0 and variance matching that of ϵ_t) or nonparametrically estimated probability density (e.g., using kernel smoothing) for incorporating additional variability in the data.

8.4. Bootstrapped tests for trend detection in time series

8.4.1. Bootstrapped t-test and Mann–Kendall test

Noguchi et al. (2011) enhanced the classical t-test and Mann–Kendall trend test with sieve bootstrap approaches that take into account the serial correlation of data to obtain more accurate and reliable estimates. While taking into account the dependence structure in the data, these tests are still limited to the linear or monotonic case:

H_0: no trend
H_1: linear trend (t-test) or monotonic trend (Mann–Kendall test)

Apply the sieve-bootstrapped tests to our time series data, using the package `funtimes`:

```
funtimes::notrend_test(Y, ar.method = "yw")
```

```
#>
#>  Sieve-bootstrap Student's t-test for a linear trend
#>
#> data:  Y
#> Student's t value = 2, p-value = 0.1
#> alternative hypothesis: linear trend.
#> sample estimates:
#> $AR_order
#> [1] 1
#>
#> $AR_coefficients
#> phi_1
#> 0.344
```

```
funtimes::notrend_test(Y, test = "MK", ar.method = "yw")
```

```
#>
#>  Sieve-bootstrap Mann--Kendall's trend test
#>
#> data:  Y
#> Mann--Kendall's tau = 0.1, p-value = 0.2
#> alternative hypothesis: monotonic trend.
#> sample estimates:
#> $AR_order
#> [1] 1
#>
#> $AR_coefficients
#> phi_1
#> 0.344
```

Notice the different p-values from the first time we applied the tests without the bootstrap.

8.4.2. Detecting non-monotonic trends

Consider a time series

$$Y_t = \mu(t) + \epsilon_t, \tag{8.1}$$

where $t = 1, \dots, n$, $\mu(t)$ is an unknown trend function, and ϵ_t is a weakly stationary time series.

We would like to test the hypotheses

H_0: $\mu(t) = f(\theta, t)$
H_1: $\mu(t) \neq f(\theta, t)$,

where the function $f(\cdot, t)$ belongs to a known family of smooth parametric functions $S = \{f(\theta, \cdot), \theta \in \Theta\}$ and Θ is a set of possible parameter values and a subset of Euclidean space.

Special cases include

- no trend (constant trend) $f(\theta, t) \equiv 0$,
- linear trend $f(\theta, t) = \theta_0 + \theta_1 t$, and
- quadratic trend $f(\theta, t) = \theta_0 + \theta_1 t + \theta_2 t^2$.

The following local regression or the local factor test statistic was developed by Wang et al. (2008) to be applied to pre-filtered observations replicating the residuals ϵ_t in Equation 8.1. The idea is to extract the hypothesized trend $f(\theta, t)$ and group residual consecutive in time into small groups. Then, apply the ANOVA F-test for these artificial groups:

$$\begin{aligned} \text{WAVK}_n = F_n &= \frac{\text{MST}}{\text{MSE}} \\ &= \frac{k_n}{n-1} \sum_{i=1}^{n} \left(\overline{V}_{i.} - \overline{V}_{..}\right)^2 \Big/ \frac{1}{n(k_n - 1)} \sum_{i=1}^{n} \sum_{j=1}^{k_n} \left(V_{ij} - \overline{V}_{i.}\right)^2, \end{aligned}$$

where MST is the treatment sum of squares, MSE is the error sum of squares, $\{V_{i1}, \ldots, V_{ik_n}\}$ is k_n pre-filtered observations in the ith group, $\overline{V}_{i.}$ is the mean of the ith group, $\overline{V}_{..}$ is the grand mean.

Both $n \to \infty$ and $k_n \to \infty$; MSE $\to$ constant. Hence, we can consider $\sqrt{n}(\text{MST} - \text{MSE})$ instead of $\sqrt{n}(F_n - 1)$.

Lyubchich et al. (2013) extended the WAVK approach:

1. Showed that the structure of time series errors can be a linear process that is allowed not to degenerate to MA(q) or AR(p), or a conditionally heteroskedastic or GARCH process.
2. Developed a data-driven bootstrap procedure to estimate the finite sample properties of the WAVK test under the unknown dependence structure.
3. Proposed to estimate the optimal size of local windows k_n by employing the nonparametric resampling m-out-of-n selection algorithm of Bickel et al. (1997).

The WAVK test is implemented in the package `funtimes` with the same sieve bootstrap as the t-test and Mann–Kendall test. It tests the null hypothesis of no trend vs. the alternative of (non)monotonic trend.

```
funtimes::notrend_test(Y, test = "WAVK", ar.method = "yw")
```

```
#>
#>  Sieve-bootstrap WAVK trend test
#>
#> data:  Y
#> WAVK test statistic = 4, moving window = 10, p-value = 0.4
#> alternative hypothesis: (non-)monotonic trend.
```

```
#> sample estimates:
#> $AR_order
#> [1] 1
#>
#> $AR_coefficients
#> phi_1
#> 0.344
```

Also, the version of the test with the hybrid bootstrap by Lyubchich et al. (2013) is available. This version allows the user to specify different alternatives.

```
# The null hypothesis is the same as above, no trend (constant trend)
funtimes::wavk_test(Y ~ 1,
                    factor.length = "adaptive.selection",
                    ar.method = "yw",
                    out = TRUE)
```

```
#>
#>  Trend test by Wang, Akritas, and Van Keilegom (bootstrap p-values)
#>
#> data:  Y
#> WAVK test statistic = 0.1, adaptively selected window = 4, p-value =
#> 0.8
#> alternative hypothesis: trend is not of the form Y ~ 1.
#> sample estimates:
#> $trend_coefficients
#> (Intercept)
#>      -0.203
#>
#> $AR_order
#> [1] 1
#>
#> $AR_coefficients
#> phi_1
#> 0.344
#>
#> $all_considered_windows
#>  Window WAVK-statistic p-value
#>       4         0.1403   0.760
#>       5        -0.0495   0.862
#>       7        -0.0955   0.826
#>      10        -0.2146   0.866
```

```
# The null hypothesis is a quadratic trend
funtimes::wavk_test(Y ~ poly(t, 2),
                    factor.length = "adaptive.selection",
                    ar.method = "yw",
                    out = TRUE)
```

```
#>
#>  Trend test by Wang, Akritas, and Van Keilegom (bootstrap p-values)
#>
#> data:  Y
#> WAVK test statistic = 4, adaptively selected window = 4, p-value = 0.01
#> alternative hypothesis: trend is not of the form Y ~ poly(t, 2).
#> sample estimates:
#> $trend_coefficients
#> (Intercept) poly(t, 2)1 poly(t, 2)2
#>      -0.203       2.152      -1.795
#>
#> $AR_order
#> [1] 0
#>
#> $AR_coefficients
#> numeric(0)
#>
#> $all_considered_windows
#>  Window WAVK-statistic p-value
#>       4           3.76   0.010
#>       5           2.97   0.028
#>       7           1.81   0.086
#>      10           0.93   0.234
```

For the application of this test to multiple time series, see Appendix C.

> **Note**
>
> Statistical test results depend on the alternative hypothesis. Sometimes the null hypothesis cannot be rejected in favor of the alternative hypothesis because the data do not match the specified alternative. For example, there are numerous examples when in Pearson correlation analysis the null hypothesis of independence cannot be rejected in favor of the alternative hypothesis of linear dependence because the underlying nonlinear dependence cannot be described as a linear relationship. See this vignette describing a few similar cases for the time series.

8.5. Unit roots

By now, we have been identifying the order of integration (if a process $X_t \sim \mathrm{I}(d)$) by looking at the time series plot of X_t and (largely) by looking at the plot of sample ACF. We differenced time series again and again until we saw a stable mean in the time series plot and a rapid (compared with linear), exponential-like decline in ACF. Here, we present a hypothesis testing approach originally suggested by Dickey and Fuller (1979) (*Dickey–Fuller test*).

Let $X_1, \ldots, X_n$ be observations from an AR(1) model:

$$X_t - \mu = \phi_1(X_{t-1} - \mu) + Z_t,$$
$$Z_t \sim \text{WN}(0, \sigma^2),$$

where $|\phi_1| < 1$ and $\mu = \text{E}X_t$. For a large sample size n, the maximum likelihood estimator $\hat{\phi}_1$ of ϕ_1 is approximately $N(\phi_1, (1-\phi_1^2)/n)$. However, for the unit root case, this approximation is not valid! Thus, do not be tempted to use the normal approximation to construct a confidence interval for ϕ_1 and check if it includes the value 1. Instead, consider a model that assumes a unit root (H_0: unit root is present) and immediately removes it by differencing:

$$\Delta X_t = X_t - X_{t-1} = \phi_0^* + \phi_1^* X_{t-1} + Z_t, \qquad Z_t \sim \text{WN}(0, \sigma^2), \tag{8.2}$$

where $\phi_0^* = \mu(1-\phi_1)$ and $\phi_1^* = \phi_1 - 1$. Let $\hat{\phi}_1^*$ be the OLS estimator of ϕ_1^*, with its standard error estimated as

$$\widehat{\text{SE}}\left(\hat{\phi}_1^*\right) = S\left(\sum_{t=2}^{n}\left(X_{t-1} - \bar{X}\right)^2\right)^{-1/2},$$

where $S^2 = \sum_{t=2}^{n}\left(\Delta X_t - \hat{\phi}_0^* - \hat{\phi}_1^* X_{t-1}\right)^2/(n-3)$ and $\bar{X}$ is the sample mean. Dickey and Fuller (1979) derived the limit distribution of the test statistic

$$\hat{\tau}_\mu = \frac{\hat{\phi}_1^*}{\widehat{\text{SE}}\left(\hat{\phi}_1^*\right)},$$

so we know the critical levels from this distribution (the 0.01, 0.05, and 0.10 quantiles are -3.43, -2.86, and -2.57, respectively) and can test the null hypothesis of $\phi_1^* = 0$ (notice the similarity with the usual t-test for significance of regression coefficients). An important thing to remember is that the H_0 here assumes a unit root (nonstationarity).

For a more general AR(p) model, statistic $\hat{\tau}_\mu$ has a similar form (the ϕ_1^* is different: $\phi_1^* = \sum_{i=1}^{p}\phi_i - 1$), and the test is then called the *Augmented Dickey–Fuller test* (ADF test). The order p can be specified in advance or selected automatically using AIC or BIC.

Another popular test for unit roots, the *Phillips–Perron test* (PP test), is built on the ADF test and considers the same null hypothesis.

Example: ADF test applied to simulated data

Simulate a time series $Y_t \sim \text{I}(2)$ and apply the rule of thumb approach of taking differences (Figure 8.3). The results in Figure 8.3 show that two differences are enough to remove the trend ($d = 2$).

```
set.seed(123)
Z <- rnorm(200)
Yt <- ts(cumsum(cumsum(Z)))
```

```
# Apply first-order (non-seasonal) differences
D1 <- diff(Yt)

# Apply first-order (non-seasonal) differences again
D2 <- diff(D1)

p1 <- forecast::autoplot(Yt) +
    ylab("Original") +
    ggtitle("Yt")
p2 <- forecast::ggAcf(Yt) +
    ggtitle("Yt")
p3 <- forecast::autoplot(D1) +
    ylab("First differences") +
    ggtitle("(1-B)Yt")
p4 <- forecast::ggAcf(D1) +
    ggtitle("(1-B)Yt")
p5 <- forecast::autoplot(D2) +
    ylab("Second differences") +
    ggtitle("(1-B)2Yt")
p6 <- forecast::ggAcf(D2) +
    ggtitle("(1-B)2Yt")
(p1 + p2) / (p3 + p4) / (p5 + p6) +
    plot_annotation(tag_levels = 'A')
```

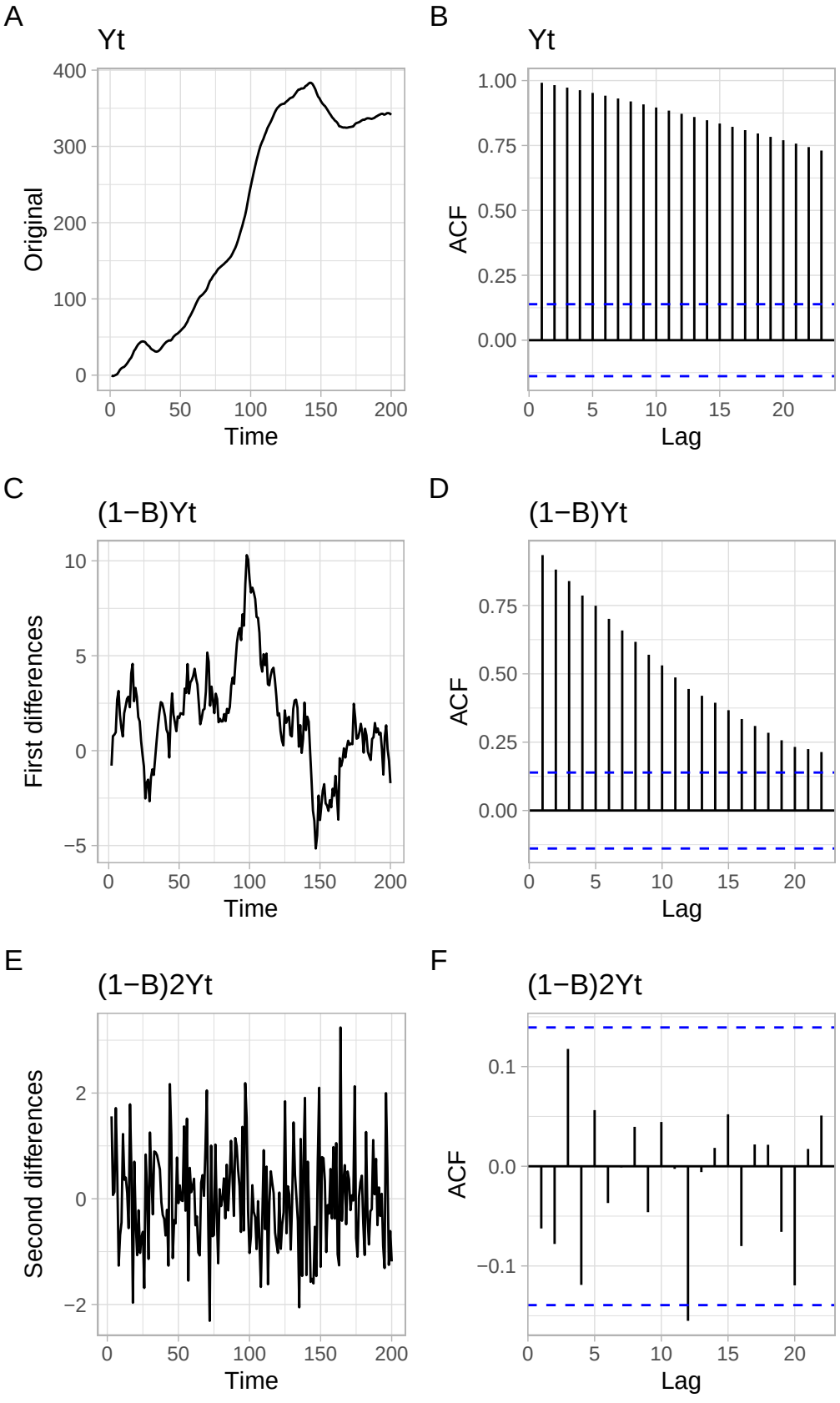


Figure 8.3.: Time series and ACF plots for identifying the order of differences d of the simulated time series.

Now apply the test, potentially several times, to identify the order d.

```
tseries::adf.test(Yt)
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  Yt
#> Dickey-Fuller = -2, Lag order = 5, p-value = 0.8
#> alternative hypothesis: stationary
```

With the high p-value, we cannot reject the H_0 of a unit root. Apply the test again on the differenced series.

```
tseries::adf.test(diff(Yt))
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  diff(Yt)
#> Dickey-Fuller = -2, Lag order = 5, p-value = 0.5
#> alternative hypothesis: stationary
```

Same result. Difference once more and re-apply the test.

```
tseries::adf.test(diff(Yt, differences = 2))
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  diff(Yt, differences = 2)
#> Dickey-Fuller = -6, Lag order = 5, p-value = 0.01
#> alternative hypothesis: stationary
```

Now, when we are using the twice differenced series $\Delta^2 Y_t = (1-B)^2 Y_t$, we can reject the H_0 and accept the alternative hypothesis of stationarity. Since the time series has been differenced twice, we state that the integration order $d = 2$, or $Y_t \sim \mathrm{I}(2)$.
What are the potential problems? Multiple testing and alternative model specifications. By model, we mean the regression Equation 8.2 that includes the parameter ϕ_1^* that we are testing. Depending on what we know or assume about the process, we may add an intercept or even a parametric trend. In R, it can be done manually or with functions from the package `urca`:

```
library(urca)
ADF <- ur.df(Yt, type = "drift", selectlags = "AIC")
summary(ADF)
```

```
#>
#> ###############################################
#> # Augmented Dickey-Fuller Test Unit Root Test #
#> ###############################################
#>
#> Test regression drift
#>
#>
#> Call:
#> lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.1896 -0.5737 -0.0834  0.5603  2.9911
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.301046   0.136667    2.20    0.029 *
#> z.lag.1     -0.000895   0.000490   -1.83    0.069 .
#> z.diff.lag   0.933704   0.024665   37.86   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.93 on 195 degrees of freedom
#> Multiple R-squared:  0.886,  Adjusted R-squared:  0.884
#> F-statistic:  755 on 2 and 195 DF,  p-value: <2e-16
#>
#>
#> Value of test-statistic is: -1.83 2.43
#>
#> Critical values for test statistics:
#>       1pct  5pct 10pct
#> tau2 -3.46 -2.88 -2.57
#> phi1  6.52  4.63  3.81
```

```
ADF <- ur.df(diff(Yt), type = "drift", selectlags = "AIC")
summary(ADF)
```

```
#>
#> ###############################################
#> # Augmented Dickey-Fuller Test Unit Root Test #
#> ###############################################
#>
#> Test regression drift
#>
#>
#> Call:
```

```
#> lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.1574 -0.5527 -0.0072  0.5857  2.9193
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   0.0812     0.0794    1.02    0.308
#> z.lag.1      -0.0536     0.0248   -2.16    0.032 *
#> z.diff.lag   -0.0362     0.0717   -0.51    0.614
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.935 on 194 degrees of freedom
#> Multiple R-squared:  0.0274, Adjusted R-squared:  0.0174
#> F-statistic: 2.73 on 2 and 194 DF,  p-value: 0.0676
#>
#>
#> Value of test-statistic is: -2.16 2.35
#>
#> Critical values for test statistics:
#>       1pct  5pct 10pct
#> tau2 -3.46 -2.88 -2.57
#> phi1  6.52  4.63  3.81
```

```
ADF <- ur.df(diff(Yt, differences = 2), type = "drift", selectlags = "AIC")
summary(ADF)
```

```
#>
#> ###############################################
#> # Augmented Dickey-Fuller Test Unit Root Test #
#> ###############################################
#>
#> Test regression drift
#>
#>
#> Call:
#> lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.1597 -0.6052 -0.0711  0.6161  3.0784
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  -0.0131     0.0675   -0.19     0.85
```

```
#> z.lag.1      -1.1534     0.1049  -10.99   <2e-16 ***
#> z.diff.lag    0.0833     0.0716    1.16     0.25
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.945 on 193 degrees of freedom
#> Multiple R-squared:  0.534,  Adjusted R-squared:  0.529
#> F-statistic:  110 on 2 and 193 DF,  p-value: <2e-16
#>
#>
#> Value of test-statistic is: -11 60.4
#>
#> Critical values for test statistics:
#>       1pct  5pct 10pct
#> tau2 -3.46 -2.88 -2.57
#> phi1  6.52  4.63  3.81
```

From the results above, the inclusion of the intercept (`"drift"`) in the test model did not affect the conclusion. Now try adding a trend (`type = "trend"` adds the intercept automatically).

```
ADF <- ur.df(Yt, type = "trend", selectlags = "AIC")
summary(ADF)
```

```
#>
#> ###############################################
#> # Augmented Dickey-Fuller Test Unit Root Test #
#> ###############################################
#>
#> Test regression trend
#>
#>
#> Call:
#> lm(formula = z.diff ~ z.lag.1 + 1 + tt + z.diff.lag)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.2098 -0.5435 -0.0667  0.5545  2.9750
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.25599    0.15611    1.64     0.10
#> z.lag.1     -0.00162    0.00131   -1.24     0.22
#> tt           0.00192    0.00320    0.60     0.55
#> z.diff.lag   0.93768    0.02558   36.66   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
```

```
#> Residual standard error: 0.932 on 194 degrees of freedom
#> Multiple R-squared:  0.886,  Adjusted R-squared:  0.884
#> F-statistic:  502 on 3 and 194 DF,  p-value: <2e-16
#>
#>
#> Value of test-statistic is: -1.24 1.73 1.85
#>
#> Critical values for test statistics:
#>       1pct  5pct 10pct
#> tau3 -3.99 -3.43 -3.13
#> phi2  6.22  4.75  4.07
#> phi3  8.43  6.49  5.47
```

```
ADF <- ur.df(diff(Yt), type = "trend", selectlags = "AIC")
summary(ADF)
```

```
#>
#> ###############################################
#> # Augmented Dickey-Fuller Test Unit Root Test #
#> ###############################################
#>
#> Test regression trend
#>
#>
#> Call:
#> lm(formula = z.diff ~ z.lag.1 + 1 + tt + z.diff.lag)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.1777 -0.5443 -0.0691  0.5697  2.9673
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.24888    0.15538    1.60    0.111
#> z.lag.1     -0.06228    0.02571   -2.42    0.016 *
#> tt          -0.00153    0.00122   -1.26    0.211
#> z.diff.lag  -0.03604    0.07155   -0.50    0.615
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.933 on 193 degrees of freedom
#> Multiple R-squared:  0.0353, Adjusted R-squared:  0.0203
#> F-statistic: 2.35 on 3 and 193 DF,  p-value: 0.0735
#>
#>
#> Value of test-statistic is: -2.42 2.1 3.13
#>
```

```
#> Critical values for test statistics:
#>       1pct  5pct 10pct
#> tau3 -3.99 -3.43 -3.13
#> phi2  6.22  4.75  4.07
#> phi3  8.43  6.49  5.47
```

```
ADF <- ur.df(diff(Yt, differences = 2), type = "trend", selectlags = "AIC")
summary(ADF)
```

```
#>
#> ###############################################
#> # Augmented Dickey-Fuller Test Unit Root Test #
#> ###############################################
#>
#> Test regression trend
#>
#>
#> Call:
#> lm(formula = z.diff ~ z.lag.1 + 1 + tt + z.diff.lag)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.1801 -0.6461 -0.0636  0.6026  3.1220
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.064035   0.136866    0.47     0.64
#> z.lag.1     -1.156959   0.105235  -10.99   <2e-16 ***
#> tt          -0.000775   0.001196   -0.65     0.52
#> z.diff.lag   0.085265   0.071776    1.19     0.24
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.946 on 192 degrees of freedom
#> Multiple R-squared:  0.535,  Adjusted R-squared:  0.527
#> F-statistic: 73.5 on 3 and 192 DF,  p-value: <2e-16
#>
#>
#> Value of test-statistic is: -11 40.3 60.4
#>
#> Critical values for test statistics:
#>       1pct  5pct 10pct
#> tau3 -3.99 -3.43 -3.13
#> phi2  6.22  4.75  4.07
#> phi3  8.43  6.49  5.47
```

In this simulated example, misspecification of the testing model did not change our conclusion

that $X_t \sim \mathrm{I}(2)$, due to the automatic adjusting of the critical values `tau` to `tau2` (model with intercept) and `tau3` (model with trend).

8.5.1. ADF and PP test problems

The ADF and PP tests are asymptotically equivalent but may differ substantially in finite samples due to the different ways in which they correct for serial correlation in the test regression.

In general, the ADF and PP tests have very low power against I(0) alternatives that are close to being I(1). That is, unit root tests cannot distinguish highly persistent stationary processes from nonstationary processes very well. Also, the power of unit root tests diminishes as deterministic terms are added to the test regressions. Tests that include a constant and trend in the test regression have less power than tests that only include a constant in the test regression.

8.6. Conclusion

Temporal dependence of time series is the most violated assumption of classical tests often employed for detecting trends, including the nonparametric Mann–Kendall test. However, multiple workarounds exist that allow the analyst to relax (avoid) the assumption of independence and provide more reliable inference. We have implemented sieve bootstrapping for time series as one such workaround.

Test results depend on the specified alternative hypothesis. Remember that non-rejection of the null hypothesis doesn't make it automatically true.

When testing for unit roots (integration) in time series, the null hypothesis of ADF and PP tests is the unit root, while the alternative is the stationarity of the tested series. Iterative testing and differencing can be used to identify the order of integration d.

Part IV.

Module 4 - Deep Learning

9. Neural Network

The objective is to build and analyze a neural network from the ground up, manually implementing all core components to understand the underlying mathematics and mechanics of training, forward propagation, and backpropagation. By constructing the network step by step, we aim to gain a deeper understanding of how neural networks learn patterns and make predictions.

Objectives

1. Understand the architecture and components of a neural network (neurons, layers, weights, activation functions).
2. Implement forward propagation and compute loss functions manually.
3. Derive and implement the backpropagation algorithm for weight updates.
4. Train and test the model on sample data to evaluate accuracy and generalization.
5. For validation and efficiency, compare manual implementation with high-level frameworks (e.g., TensorFlow, PyTorch).

Reading materials

- Chapter 4 in (**Goodfellow:Bengio:Courville:2016?**) — Deep Learning
- Section 2.3 in (**Bishop:2006?**) — Pattern Recognition and Machine Learning
- Online Tutorial: "Neural Networks from Scratch in Python" by Harrison Kinsley & Daniel Kukieła (2020)

9.1. Introduction

In contrast to the traditional time series analysis that focuses on modeling the conditional first moment, models of *autoregressive conditional heteroskedasticity* (ARCH) and *generalized autoregressive conditional heteroskedasticity* (GARCH) specifically take the dependency of the conditional second moment into modeling consideration and accommodate the increasingly important need to explain and model risk and uncertainty in, for example, financial time series.

The ARCH models were introduced in 1982 by Robert Engle to model varying (conditional) variance or volatility of time series. It is often found in economics that the larger values of time series also lead to larger instability (i.e., larger variances), which is termed *(conditional) heteroskedasticity*. Standard examples for demonstrating ARCH or GARCH effects are time series of stock prices, interest and foreign exchange rates, and even some environmental processes: high-frequency data on wind speed, energy production, air quality, etc. (see examples in Cripps and Dunsmuir 2003; Marinova and McAleer 2003; Taylor and Buizza 2004; Campbell and Diebold 2005). In 2003, Robert F. Engle was awarded 1/2 of the Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel for his work on ARCH models (the other half was awarded to C. Granger, see Section 11.3). Although it is not one of the prizes that Alfred Nobel established in his will in 1895, the Sveriges Riksbank Prize is referred to along with the other Nobel Prizes by the Nobel Foundation.

9.2. Features of ARCH

Since financial data typically have the autocorrelation coefficient close to 1 at lag 1 (e.g., the exchange rate between the US and Canadian dollar hardly changes from today to tomorrow), it is much more interesting and also more practically relevant to model the returns of a financial time series rather than the series itself. Let Y_t be a time series of stock prices. The returns X_t measure the relative changes in price and are typically defined as simple returns

$$X_t = \frac{Y_t - Y_{t-1}}{Y_{t-1}} = \frac{Y_t}{Y_{t-1}} - 1 \tag{9.1}$$

or logarithmic returns

$$X_t = \ln Y_t - \ln Y_{t-1}. \tag{9.2}$$

The two forms are approximately the same, since

$$\begin{aligned} \ln Y_t - \ln Y_{t-1} &= \ln\left(\frac{Y_t}{Y_{t-1}}\right) \\ &= \ln\left(\frac{Y_{t-1} + Y_t - Y_{t-1}}{Y_{t-1}}\right) \\ &= \ln\left(1 + \frac{Y_t - Y_{t-1}}{Y_{t-1}}\right) \\ &\approx \frac{Y_t - Y_{t-1}}{Y_{t-1}}. \end{aligned} \tag{9.3}$$

The approximation $\ln(1+x) \approx x$ works when x is close to zero, which is true for many real-world financial problems. However, logarithmic returns are often preferred because in many applications their distribution is closer to normal compared to one of simple returns. Also, log returns have infinite support (from $-\infty$ to $+\infty$) compared to simple returns that have a lower bound of -1.

For an example calculation of log returns, see Figure 9.1.

```
# Load data and calculate log returns
CAD <- readr::read_csv("data/CAD.csv",
                       na = "Bank holiday",
                       skip = 11) %>%
    filter(!is.na(USD)) %>%
    mutate(lnR = c(NA, diff(log(USD)))) %>%
    filter(!is.na(lnR))

p1 <- ggplot(CAD, aes(x = Date, y = USD)) +
    geom_line() +
    ylab("CAD per USD")
p2 <- ggplot(CAD, aes(x = Date, y = lnR)) +
    geom_line() +
    ylab("Log return")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

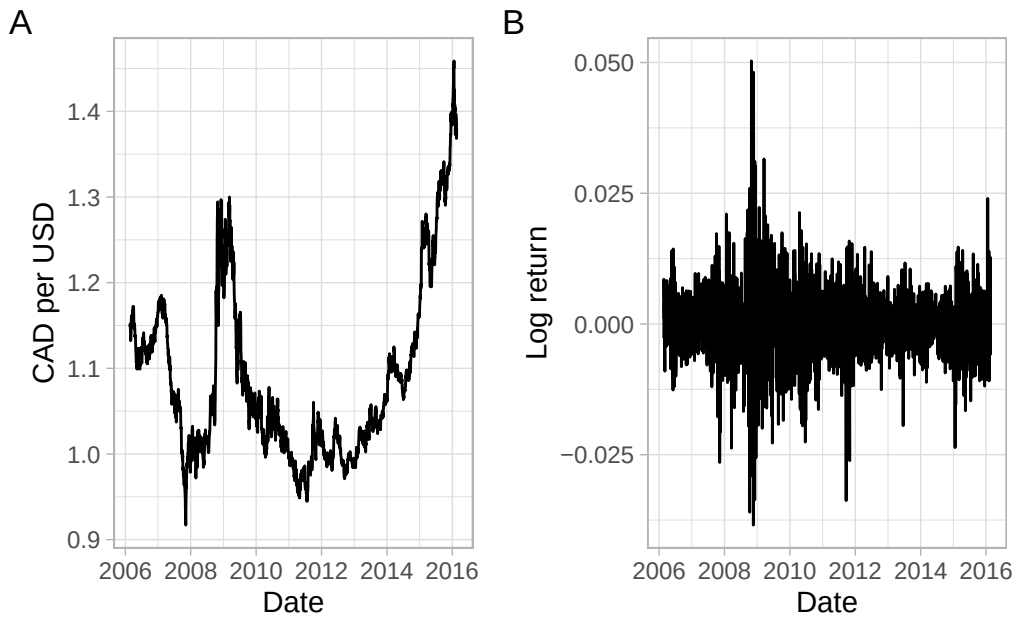


Figure 9.1.: CAD per USD daily noon exchange rates and log returns, from 2006-02-22 to 2016-02-22 (excluding bank holidays), obtained from Bank of Canada.

Rydberg (2000) summarizes some important *stylized features* of financial return series, which have been repeatedly observed in all kinds of assets including stock prices, interest rates, and foreign exchange rates:

1. *Heavy tails.* The distribution of the returns X_t has tails heavier than the tails of a normal distribution.
2. *Volatility clustering.* Large price changes occur in clusters. Indeed, often large price changes tend to be followed by large price changes, and periods of tranquility alternate with periods of high volatility.
3. *Asymmetry.* There is evidence that the distribution of stock returns is slightly negatively skewed. One possible explanation could be that trades react more strongly to negative information than to positive information.
4. *Aggregational Gaussianity.* When the sampling frequency decreases, the central limit law sets in and the distribution of the returns over a long period tends toward a normal distribution.
5. *Long-range dependence.* The returns themselves hardly show any autocorrelation, which, however, does not mean that they are independent. Both squared returns and absolute returns often exhibit persistent autocorrelations indicating possible long-memory dependence in those series.

Figure 9.2 is the simplest check for the presence of ARCH effects: when the time series is not autocorrelated but is autocorrelated if squared.

```
p1 <- forecast::ggAcf(CAD$lnR) +
    ggtitle("Log returns")
p2 <- forecast::ggAcf(CAD$lnR^2) +
```

```
    ggtitle(bquote('(Log returns)'^2))
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

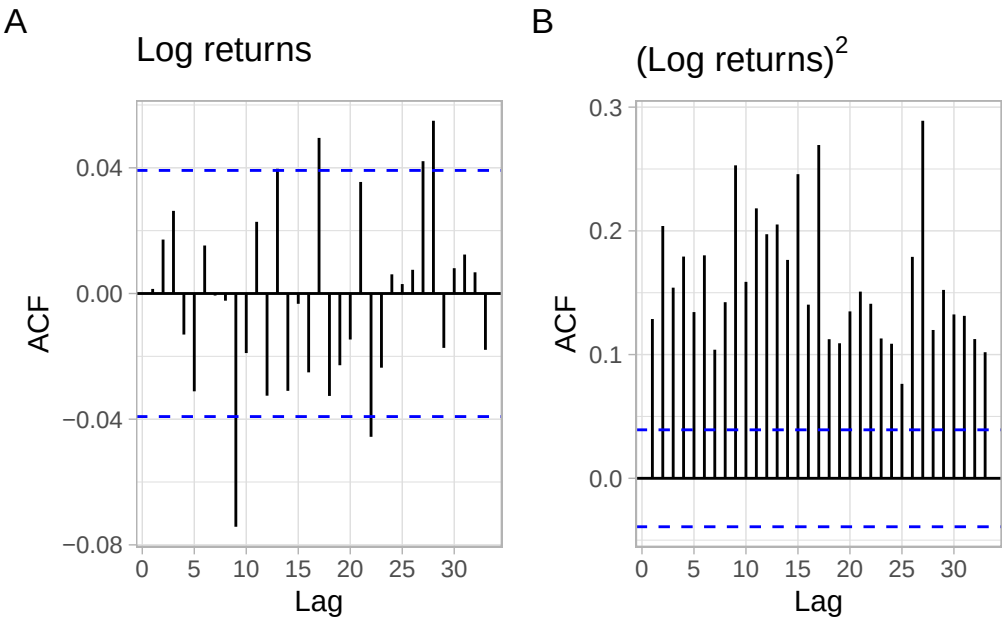


Figure 9.2.: ACF of log returns and squared log returns for USD/CAD exchange.

For comparison, see Figure 9.3 with similar plots for simulated i.i.d. series.

```
set.seed(1)
iid <- rnorm(nrow(CAD))

p1 <- forecast::ggAcf(iid) +
    ggtitle("iid")
p2 <- forecast::ggAcf(iid^2) +
    ggtitle(bquote('iid'^2))
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

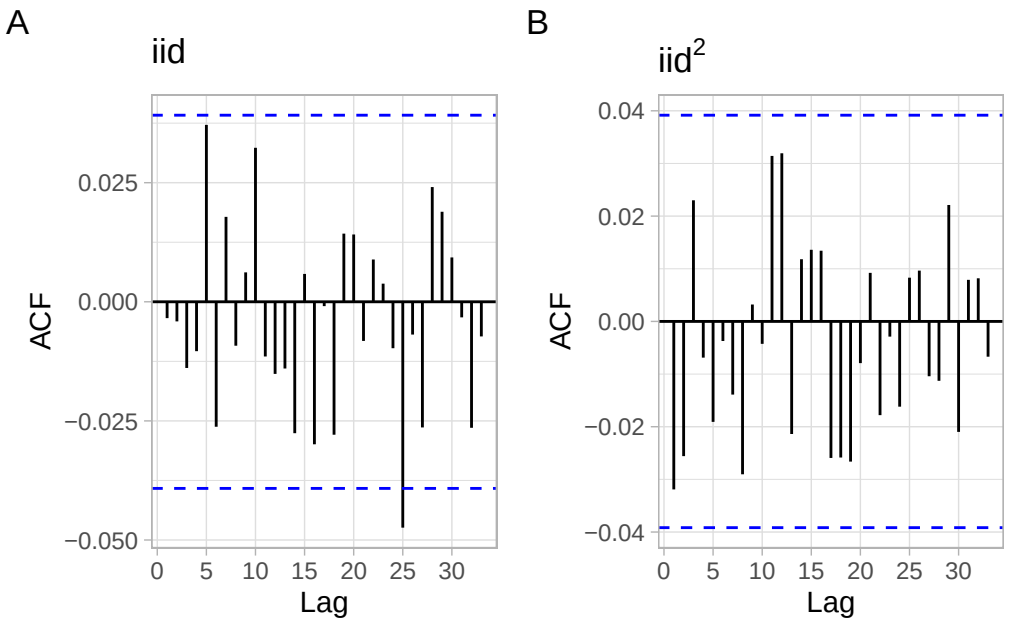


Figure 9.3.: ACF of simulated i.i.d. $N(0,1)$ series.

9.3. Models

Engle (1982) defines an autoregressive conditional heteroskedastic (ARCH) model as

$$\begin{aligned} X_t &= \sigma_t \varepsilon_t, \\ \sigma_t^2 &= a_0 + a_1 X_{t-1}^2 + \cdots + a_p X_{t-p}^2, \end{aligned} \tag{9.4}$$

where $a_0 > 0$, $a_j \geqslant 0$, $\varepsilon_t \sim$ i.i.d.$(0,1)$, and ε_t is independent of X_{t-j}, where $j \geqslant 1$. We write $X_t \sim \text{ARCH}(p)$.

We can see that

$$\begin{aligned} \mathrm{E}X_t &= 0, \\ \operatorname{var}\left(X_t | X_{t-1}, \ldots, X_{t-p}\right) &= \sigma_t^2, \\ \operatorname{cov}\left(X_t, X_k\right) &= 0 \quad \text{for all} \quad t \neq k. \end{aligned}$$

> **Note**
>
> Stationary ARCH is white noise.

Thus, in ARCH, the predictive distribution of X_t based on its past is a scale-transform of the distribution of ε_t with the scaling constant σ_t depending on the past of the process.

Bollerslev (1986) introduced a generalized autoregressive conditional heteroskedastic (GARCH) model by replacing the second formula in Equation 9.4 with

$$\begin{aligned}\sigma_t^2 &= a_0 + a_1 X_{t-1}^2 + \cdots + a_p X_{t-p}^2 + b_1 \sigma_{t-1}^2 + \cdots + b_q \sigma_{t-q}^2 \\ &= a_0 + \sum_{i=1}^{p} a_i X_{t-i}^2 + \sum_{j=1}^{q} b_j \sigma_{t-j}^2,\end{aligned} \tag{9.5}$$

where $a_0 > 0$, $a_i \geqslant 0$, and $b_j \geqslant 0$. We write $X_t \sim \text{GARCH}(p, q)$.

Notice the similarity between ARMA and GARCH models.

The parameters of ARCH/GARCH models are estimated using the method of conditional maximum likelihood. There exist several tests for ARCH/GARCH effects (e.g., analyzing time series and ACF plots, the Engle's Lagrange multiplier test).

The approaches to selecting the orders p and q for GARCH include:

- Visual analysis of ACF and PACF of the squared time series and other residual diagnostics;
- Variations of information criteria such as AIC and BIC to account for the number of estimated parameters in the GARCH model (Brooks and Burke 2003);
- Using GARCH(1,1) by following Hansen and Lunde (2005);
- Using out-of-sample forecasts (comparing alternative model specifications on a testing set).

9.3.1. Lagrange multiplier test

The Lagrange multiplier (LM) test is equivalent to an F-test for the significance of the least squares regression on squared values:

$$X_t^2 = \alpha_0 + \alpha_1 X_{t-1}^2 + \cdots + \alpha_m X_{t-m}^2 + e_t, \tag{9.6}$$

where e_t denotes the error term, m is a positive integer, $t = m+1, \ldots, T$, and T is the sample size (length of the time series).

Specifically, the null hypothesis is

$$H_0 : \alpha_1 = \cdots = \alpha_m = 0.$$

Let the sum of squares total

$$SST = \sum_{t=m+1}^{T} \left(X_t^2 - \overline{X_t^2} \right)^2,$$

where $\overline{X_t^2}$ is the sample mean of X_t^2. The sum of squares of the errors

$$SSE = \sum_{t=m+1}^{T} \hat{e}_t^2,$$

where $\hat{e}_t$ is the least-squares residual of the linear regression (Equation 9.6).

Then, the test statistic

$$F = \frac{(SST - SSE)/m}{SSE/(T - 2m - 1)}, \tag{9.7}$$

which is asymptotically distributed under the null hypothesis as a χ^2 distribution with m degrees of freedom.

Example: Testing ARCH effects in USD/CAD log returns

We have seen in Figure 9.2 the autocorrelation of squared log returns. Now apply the formal LM test.

```
m <- 12
FinTS::ArchTest(CAD$lnR, lags = m)
```

```
#>
#>  ARCH LM-test; Null hypothesis: no ARCH effects
#>
#> data:  CAD$lnR
#> Chi-squared = 377, df = 12, p-value <2e-16
```

The LM test implemented in the function `FinTS::ArchTest()` detects the presence of ARCH effects (rejects the null hypothesis) when considering 12 lags. In base R, the same results can be obtained by running the F test as follows:

```
mat <- embed(CAD$lnR^2, m + 1)
mod <- lm(mat[,1] ~ mat[,-1])
anova(mod)
```

```
#> Analysis of Variance Table
#>
#> Response: mat[, 1]
#>             Df   Sum Sq  Mean Sq F value Pr(>F)
#> mat[, -1]   12 5.15e-06 4.29e-07    36.8 <2e-16 ***
#> Residuals 2479 2.89e-05 1.20e-08
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Example: GARCH model for USD/CAD log returns

Let's estimate a GARCH(1,1) model for these data, using the conditional ML method. Note that `omega` in the results below is denoted a_0 in our equations (the intercept in the variance model):

```
library(fGarch)
garch11 <- fGarch::garchFit(lnR ~ garch(1, 1),
                            data = CAD, trace = FALSE)
garch11@description <- "---"
garch11
```

```
#>
#> Title:
#>  GARCH Modelling
#>
```

```
#> Call:
#>  fGarch::garchFit(formula = lnR ~ garch(1, 1), data = CAD, trace = FALSE)
#>
#> Mean and Variance Equation:
#>  lnR ~ garch(1, 1)
#>  [data = CAD]
#>
#> Conditional Distribution:
#>  norm
#>
#> Coefficient(s):
#>          mu        omega       alpha1        beta1
#> -4.5267e-05   2.2304e-07   5.6511e-02   9.3805e-01
#>
#> Std. Errors:
#>  based on Hessian
#>
#> Error Analysis:
#>         Estimate  Std. Error  t value Pr(>|t|)
#> mu     -4.527e-05   9.863e-05   -0.459  0.64628
#> omega   2.230e-07   6.893e-08    3.236  0.00121 **
#> alpha1  5.651e-02   6.699e-03    8.436  < 2e-16 ***
#> beta1   9.381e-01   6.846e-03  137.013  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Log Likelihood:
#>  9445    normalized:  3.77
#>
#> Description:
#>  ---
```

Next, we run diagnostics of the fitted model, i.e., whether the residuals ε_t are white noise and are normally distributed. The code `plot(garch11)` generates scatterplots, histograms, Q-Q plots, and ACF plots of the original data and the obtained residuals (Figure 9.4). Of course, the analysis can be performed also with separate commands. For example, see the ACFs of the residuals (Figure 9.5).

```
par(mfrow = c(1, 2))
plot(garch11, which = c(3, 13))
```

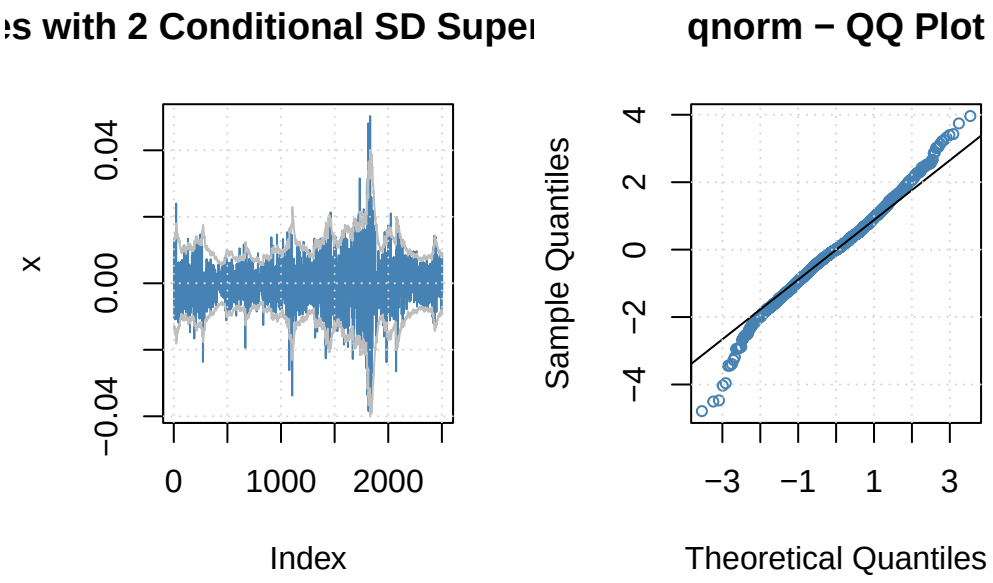


Figure 9.4.: Selected diagnostics of the GARCH(1,1) model for the USD/CAD log returns.

```
et <- residuals(garch11, standardize = TRUE)
p1 <- forecast::ggAcf(et) +
    ggtitle("Residuals")
p2 <- forecast::ggAcf(et^2) +
    ggtitle(bquote('Residuals'^2))
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

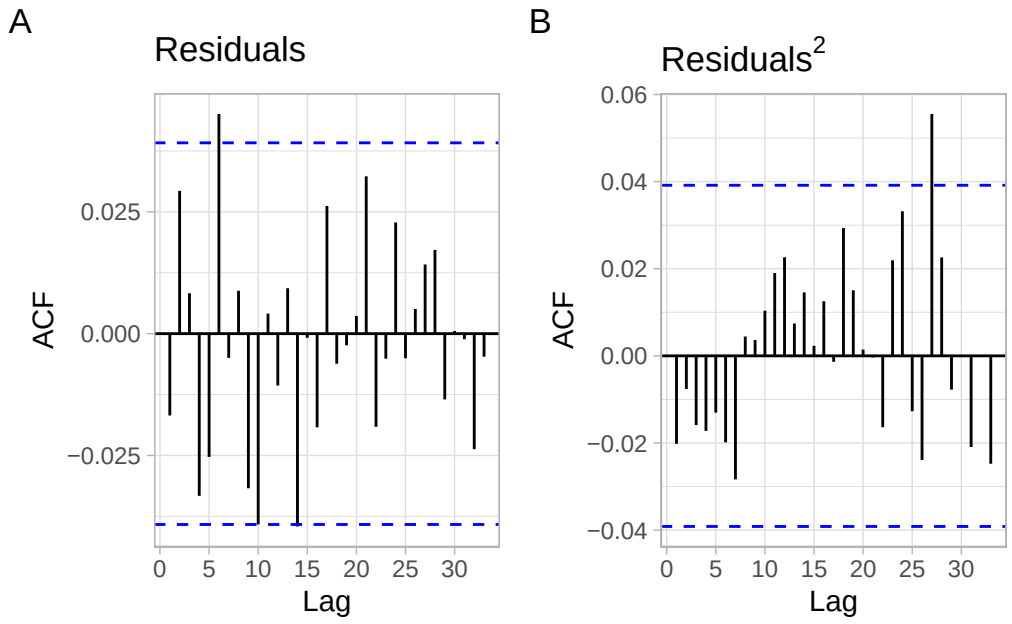


Figure 9.5.: ACF of residuals from the GARCH(1,1) model for the USD/CAD log returns.

Based on the plots, autocorrelations were effectively removed, but the assumption of normality is violated – Figure 9.4 shows heavy and almost symmetric tails in the distribution of GARCH residuals.
To account for the heavy tails, we change the conditional distribution from normal to the standardized Student t-distribution (see `?fGarch::std`).

```
garch11t <- fGarch::garchFit(lnR ~ garch(1, 1),
                             data = CAD, trace = FALSE,
                             cond.dist = "std")
garch11t@description <- "---"
garch11t
```

```
#>
#> Title:
#>  GARCH Modelling
#>
#> Call:
#>  fGarch::garchFit(formula = lnR ~ garch(1, 1), data = CAD, cond.dist = "std",
#>     trace = FALSE)
#>
#> Mean and Variance Equation:
#>  lnR ~ garch(1, 1)
#>  [data = CAD]
#>
#> Conditional Distribution:
```

```
#>  std
#>
#> Coefficient(s):
#>          mu         omega        alpha1         beta1         shape
#> -3.1968e-05   1.8744e-07   5.5352e-02   9.4057e-01   8.6423e+00
#>
#> Std. Errors:
#>  based on Hessian
#>
#> Error Analysis:
#>          Estimate  Std. Error  t value Pr(>|t|)
#> mu     -3.197e-05   9.433e-05   -0.339   0.7347
#> omega   1.874e-07   8.038e-08    2.332   0.0197 *
#> alpha1  5.535e-02   7.765e-03    7.128 1.02e-12 ***
#> beta1   9.406e-01   7.760e-03  121.210  < 2e-16 ***
#> shape   8.642e+00   1.421e+00    6.084 1.17e-09 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Log Likelihood:
#>  9473    normalized:  3.78
#>
#> Description:
#>  ---
```

Verify the diagnostics plots in Figure 9.6.

```
par(mfrow = c(2, 2))
plot(garch11t, which = c(3, 13))
plot(garch11t, which = c(9, 11))
```

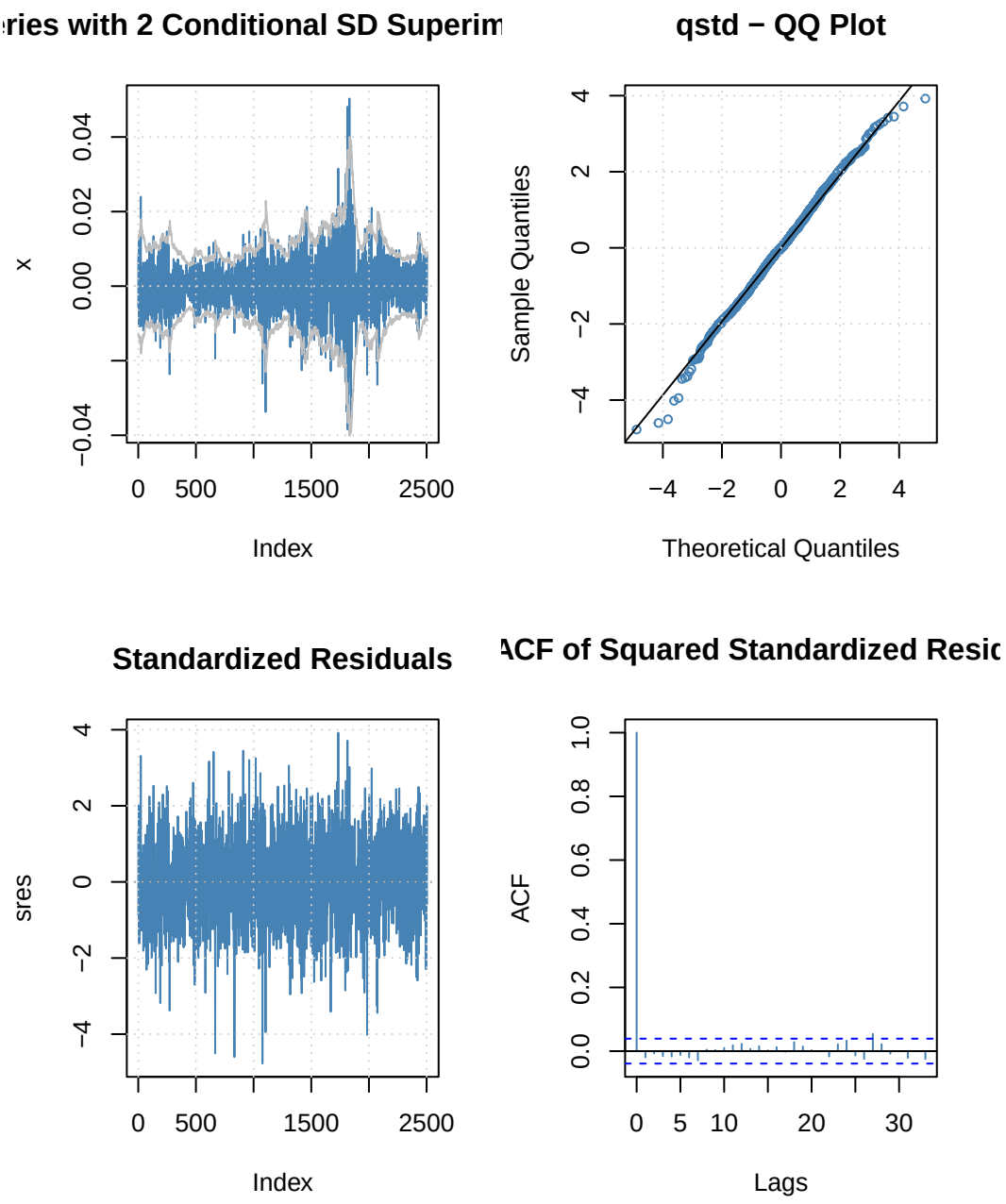


Figure 9.6.: Selected diagnostics of the GARCH(1,1) model with the standardized Student-t distribution for the USD/CAD log returns.

Now we can use the model for predictions (Figure 9.7). Few things to note here:

- the model is 'self-contained' so we can just specify `n.ahead` for the number of steps to be predicted;
- GARCH models are typically applied to long time series, so the argument `nx` limits the length of the plotted observed time series (by default only the most recent 25% of

observations are plotted);
- the argument `conf` specifies the confidence level, then the conditional distribution from the model is used to compute critical values for the interval forecasts. Alternatively, one can specify the critical values manually with the argument `crit_val`.

For more details, see `?fGarch::predict`.

```
predict(garch11t, n.ahead = 30,
        conf = 0.95,
        plot = TRUE)
```

```
#>    meanForecast meanError standardDeviation lowerInterval upperInterval
#> 1      -3.2e-05   0.00445           0.00445      -0.00892       0.00885
#> 2      -3.2e-05   0.00446           0.00446      -0.00894       0.00888
#> 3      -3.2e-05   0.00448           0.00448      -0.00896       0.00890
#> 4      -3.2e-05   0.00449           0.00449      -0.00899       0.00892
#> 5      -3.2e-05   0.00450           0.00450      -0.00901       0.00895
#> 6      -3.2e-05   0.00451           0.00451      -0.00903       0.00897
#> 7      -3.2e-05   0.00452           0.00452      -0.00906       0.00899
#> 8      -3.2e-05   0.00453           0.00453      -0.00908       0.00902
#> 9      -3.2e-05   0.00455           0.00455      -0.00910       0.00904
#> 10     -3.2e-05   0.00456           0.00456      -0.00913       0.00906
#> 11     -3.2e-05   0.00457           0.00457      -0.00915       0.00908
#> 12     -3.2e-05   0.00458           0.00458      -0.00917       0.00911
#> 13     -3.2e-05   0.00459           0.00459      -0.00919       0.00913
#> 14     -3.2e-05   0.00460           0.00460      -0.00921       0.00915
#> 15     -3.2e-05   0.00461           0.00461      -0.00924       0.00917
#> 16     -3.2e-05   0.00462           0.00462      -0.00926       0.00919
#> 17     -3.2e-05   0.00463           0.00463      -0.00928       0.00922
#> 18     -3.2e-05   0.00464           0.00464      -0.00930       0.00924
#> 19     -3.2e-05   0.00466           0.00466      -0.00932       0.00926
#> 20     -3.2e-05   0.00467           0.00467      -0.00934       0.00928
#> 21     -3.2e-05   0.00468           0.00468      -0.00937       0.00930
#> 22     -3.2e-05   0.00469           0.00469      -0.00939       0.00932
#> 23     -3.2e-05   0.00470           0.00470      -0.00941       0.00934
#> 24     -3.2e-05   0.00471           0.00471      -0.00943       0.00936
#> 25     -3.2e-05   0.00472           0.00472      -0.00945       0.00938
#> 26     -3.2e-05   0.00473           0.00473      -0.00947       0.00940
#> 27     -3.2e-05   0.00474           0.00474      -0.00949       0.00942
#> 28     -3.2e-05   0.00475           0.00475      -0.00951       0.00945
#> 29     -3.2e-05   0.00476           0.00476      -0.00953       0.00947
#> 30     -3.2e-05   0.00477           0.00477      -0.00955       0.00949
```

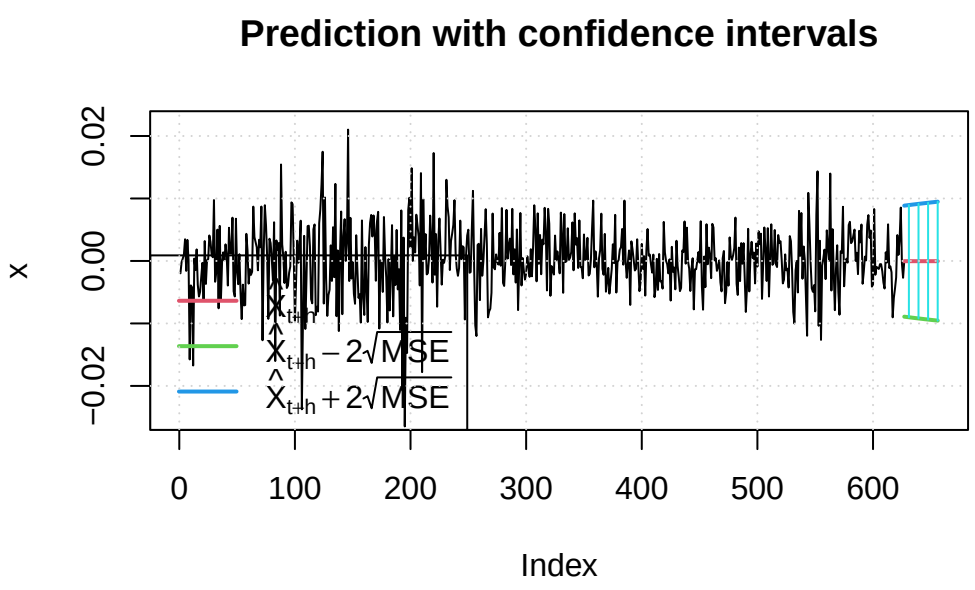


Figure 9.7.: Predictions using the GARCH(1,1) model for the USD/CAD log returns.

9.4. Extensions

There was a boom in creating new models by adding new features to GARCH:

- IGARCH – integrated GARCH
- EGARCH – exponential GARCH
- TGARCH – threshold GARCH
- QGARCH – quadratic GARCH
- GARCH-M – GARCH with heteroskedasticity in mean
- NGARCH – nonlinear GARCH
- …
- MARCH – modified GARCH
- STARCH – structural ARCH
- …

Thus, these papers had to appear: Hansen and Lunde (2005) and Bollerslev (2009).

9.5. Model building

We have considered the models for conditional heteroskedasticity, and in the example, we estimated the mean just as a constant (intercept `mu`), however, in a more general case one might need to model and remove trend and cyclical variability along with autocorrelations (recall the methods of smoothing, ARMA, and ARIMA modeling) before exploring the need to model ARCH.

Below are the steps for such a more general case of analysis, adapted from Chapter 3.3 in Tsay (2005):

1. Specify a mean equation by testing for trend and serial dependence in the data and, if necessary, build a time series model (e.g., an ARMA model) to remove any linear dependence.
2. Use the residuals of the mean equation to test for ARCH effects.
3. If the ARCH effects are statistically significant, specify a volatility model and perform a *joint estimation* of the mean and volatility equations.
4. Check the fitted model carefully and refine it if necessary.

> **Note**
>
> The joint estimation can be done in R using the function `fGarch::garchFit()` and specifying, e.g., `formula = ~ arma(2, 1) + garch(1, 1)`.

9.6. Conclusion

Whereas originated in financial analysis, GARCH models are becoming popular in other domains, including environmental science. Note that close alternatives exist, for example, GAMLSS allows modeling different distributional parameters such as mean, scale, skewness, etc.

GARCH effects are tested for and modeled *after* the mean (i.e., trend) and autocorrelations are removed. Standard model selection techniques can be adapted to specify GARCH models.

R packages offering functions for GARCH modeling include (in alphabetic order): `bayesforecast`, `betategarch`, `fGarch` (used here), `garchx`, `rmgarch`, `rugarch`, `tseries`, and more (see, for example, CRAN Task Views on Empirical Finance).

10. Computer Vision and Visual Intelligence

The goal of this lecture is to learn how to view time series from the frequency perspective. You should be able to recognize features of time series from a periodogram similar to how we did it using plots of the autocorrelation function (ACF).

Objectives

1. Describe the mechanics of Fourier transform for time series.
2. Present results of spectrum calculations using a periodogram or spectrogram.
3. Give examples of smoothed periodograms.
4. Demonstrate the use of the Lomb–Scargle periodogram for analysis of unequally-spaced time series.

Reading materials

- Chapter 4 in Shumway and Stoffer (2017)
- Nason (2008) on wavelet analysis

10.1. Introduction

By now, we have been working in the *time domain*, such that our analysis could be seen as a regression of the present on the past (for example, ARIMA models). We have been using many time series plots with time on the x-axis.

An alternative approach is to analyze time series in a *spectral domain*, such that use a regression of present on a linear combination of sine and cosine functions. In this type of analysis, we often use periodogram plots, with frequency or period on the x-axis.

10.2. Regression on sinusoidal components

The simplest form of spectral analysis consists of regression on a periodic component:

$$Y_t = A\cos\omega t + B\sin\omega t + C + \epsilon_t, \tag{10.1}$$

where $t = 1, \dots, T$ and $\epsilon_t \sim \text{WN}(0, \sigma^2)$. Without loss of generality, we assume $0 \leqslant \omega \leqslant \pi$. In fact, for discrete data, frequencies outside this range are aliased into this range. For example, suppose that $-\pi < (\omega = \pi - \delta) < 0$, then

$$\begin{aligned}\cos(\omega t) &= \cos((\pi - \delta)t) \\ &= \cos(\pi t)\cos(\delta t) + \sin(\pi t)\sin(\delta t) \\ &= \cos(\delta t).\end{aligned}$$

Hence, a sampled sinusoid with a frequency smaller than 0 appears to coincide with a sinusoid with frequency in the interval $[0, \pi]$.

> **Note**
>
> The term $A\cos\omega t + B\sin\omega t$ is a periodic function with the period $2\pi/\omega$. The period $2\pi/\omega$ represents the number of time units that it takes for the function to take the same value again, i.e., to complete a cycle. The frequency, measured in cycles per time unit, is given by the inverse $\omega/(2\pi)$. The angular frequency, measured in radians per time unit, is given by ω. Because of its convenience, the angular frequency ω will be used to describe the periodicity of the function, and its name is shortened to frequency when there is no danger of confusion.

Consider monthly data that exhibit a 12-month seasonality. Hence, the period $2\pi/\omega = 12$, which implies the angular frequency $\omega = \pi/6$. The frequency, measured in cycles per time unit, is given by the inverse

$$\frac{\omega}{2\pi} = \frac{1}{12} \approx 0.08.$$

The formulas to estimate parameters of regression in Equation 19.1 take a much simpler form if ω is one of the Fourier frequencies, defined by

$$\omega_j = \frac{2\pi j}{T}, \quad j = 0, \ldots, \frac{T}{2},$$

then

$$\begin{aligned}
\hat{A} &= \frac{2}{T}\sum_t Y_t \cos\omega_j t, \\
\hat{B} &= \frac{2}{T}\sum_t Y_t \sin\omega_j t, \\
\hat{C} &= \overline{Y} = \frac{1}{T}\sum_t Y_t.
\end{aligned}$$

A suitable way of testing the significance of the sinusoidal component with frequency ω_j is using its contribution to the sum of squares

$$R_T(\omega_j) = \frac{T}{2}\left(\hat{A}^2 + \hat{B}^2\right).$$

If the $\epsilon_t \sim N(0, \sigma^2)$, then it follows that $\hat{A}$ and $\hat{B}$ are also independent normal, each with the variance $2\sigma^2/T$, so under the null hypothesis of $A = B = 0$ we find that

$$\frac{R_T(\omega_j)}{\sigma^2} \sim \chi^2_2$$

or equivalently that $R_T(\omega_j)/(2\sigma^2)$ has an exponential distribution with mean 1. The above theory can be extended to the simultaneous estimation of several periodic components.

10.3. Periodogram

The *Fourier transform* uses Fourier series, such as the pairs of sines and cosines with different periods, to describe the frequencies present in the original time series.

The Fourier transform applied to an equally-spaced time series Y_t (where $t = 1, \dots, T$) is also called the *discrete Fourier transform* (DFT) because the time is discrete (not the values Y_t).

To reduce the computational complexity of DFT and speed up the computations, one of the *fast Fourier transform* (FFT) algorithms is typically used.

The results of the Fourier transform are shown in a periodogram that describes the spectral properties of the signal.

The periodogram is defined as

$$I_T(\omega) = \frac{1}{2\pi T} \left| \sum_{t=1}^{T} Y_t e^{i\omega t} \right|^2 ,$$

which is an approximately unbiased estimator of the spectral density f.

Some undesirable features of the periodogram:

1. $I_T(\omega)$ for fixed ω is not a consistent estimate of $f(\omega)$, since

$$I_T(\omega_j) \sim \frac{f(\omega_j)}{2} \chi_2^2.$$

   Therefore, the variance of $f^2(\omega)$ does not tend to 0 as $T \to \infty$.
2. The independence of periodogram ordinates at different Fourier frequencies suggests that the sample periodogram plotted as a function of ω will be extremely irregular.

Suppose that $\gamma(h)$ is the autocovariance function of a stationary process and that $f(\omega)$ is the spectral density for the same process (h is the time lag and ω is the frequency). The autocovariance $\gamma(h)$ and the spectral density $f(\omega)$ are related:

$$\gamma(h) = \int_{-1/2}^{1/2} e^{2\pi i \omega h} f(\omega) d\omega,$$

and

$$f(\omega) = \sum_{h=-\infty}^{+\infty} \gamma(h) e^{-2\pi i \omega h}.$$

In the language of advanced calculus, the autocovariance and spectral density are Fourier transform pairs. These Fourier transform equations show that there is a direct link between the time domain representation and the frequency domain representation of a time series.

10.4. Smoothing

The idea behind smoothing is to take weighted averages over neighboring frequencies to reduce the variability associated with individual periodogram values. However, such an operation necessarily introduces some bias into the estimation procedure. Theoretical studies focus on the amount of smoothing that is required to obtain an optimum trade-off between bias and variance. In practice, this usually means that the choice of a kernel and amount of smoothing is somewhat subjective.

The main form of a smoothed estimator is given by

$$\hat{f}(\lambda) = \int_{-\pi}^{\pi} \frac{1}{h} K\left(\frac{\omega - \lambda}{h}\right) I_T(\omega) d\omega,$$

where $I_T(\cdot)$ is the periodogram based on T observations, $K(\cdot)$ is a kernel function, and h is the bandwidth. We usually take $K(\cdot)$ to be a nonnegative function, symmetric about 0 and integrating to 1. Thus, any symmetric density, such as the normal density, will work. In practice, however, it is more usual to take a kernel of finite range, such as the *Epanechnikov kernel*

$$K(x) = \frac{3}{4\sqrt{5}}\left(1 - \frac{x^2}{5}\right), \quad -\sqrt{5} \leqslant x \leqslant \sqrt{5},$$

which is 0 outside the interval $[-\sqrt{5}, \sqrt{5}]$. This choice of kernel function has some optimality properties. However, in practice, this optimality is less important than the choice of bandwidth h, which effectively controls the range over which the periodogram is smoothed.

There are some additional difficulties with the performance of the sample periodogram in the presence of a sinusoidal variation whose frequency is not one of the Fourier frequencies. This effect is known as *leakage*. The reason of leakage is that we always consider a truncated periodogram. Truncation implicitly assumes that the time series is periodic with a period T, which, of course, is not always true. So we artificially introduce non-existent periodicities into the estimated spectrum, i.e., cause 'leakage' of the spectrum. (If the time series is perfectly periodic over T then there is no leakage.) The leakage can be treated using an operation of tapering on the periodogram, i.e., by choosing appropriate periodogram windows.

> **Note**
>
> When we work with periodograms, we lose all phase (relative location/time origin) information: the periodogram will be the same if all the data were circularly rotated to a new time origin, i.e., the observed data are treated as perfectly periodic.

Another popular choice to smooth a periodogram is the *Daniell kernel* (with parameter m). For a time series, it is a centered moving average of values between the times $t - m$ and $t + m$ (inclusive). For example, the smoothing formula for a Daniell kernel with $m = 2$ is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2}}{5} \\ &= 0.2x_{t-2} + 0.2x_{t-1} + 0.2x_t + 0.2x_{t+1} + 0.2x_{t+2}.\end{aligned}$$

The weighting coefficients for a Daniell kernel with $m = 2$ can be checked using the function `kernel()`. In the output, the indices in `coef[ ]` refer to the time difference from the center of the average at the time t.

```
kernel("daniell", m = 2)
```

```
#> Daniell(2)
#> coef[-2] = 0.2
#> coef[-1] = 0.2
#> coef[ 0] = 0.2
#> coef[ 1] = 0.2
#> coef[ 2] = 0.2
```

The modified Daniell kernel is such that the two endpoints in the averaging receive half the weight that the interior points do. For a modified Daniell kernel with $m = 2$, the smoothing is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + 2x_{t-1} + 2x_t + 2x_{t+1} + x_{t+2}}{8} \\ &= 0.125x_{t-2} + 0.25x_{t-1} + 0.25x_t + 0.25x_{t+1} + 0.125x_{t+2}\end{aligned}$$

List the weighting coefficients:

```
kernel("modified.daniell", m = 2)
```

```
#> mDaniell(2)
#> coef[-2] = 0.125
#> coef[-1] = 0.250
#> coef[ 0] = 0.250
#> coef[ 1] = 0.250
#> coef[ 2] = 0.125
```

Either the Daniell kernel or the modified Daniell kernel can be convoluted (repeated) so that the smoothing is applied again to the smoothed values. This produces a more extensive smoothing by averaging over a wider time interval. For instance, to repeat a Daniell kernel with $m = 2$ on the smoothed values that resulted from a Daniell kernel with $m = 2$, the formula would be

$$\hat{\hat{x}}_t = \frac{\hat{x}_{t-2} + \hat{x}_{t-1} + \hat{x}_t + \hat{x}_{t+1} + \hat{x}_{t+2}}{5}.$$

The weights for averaging the original data for convoluted Daniell kernels with $m = 2$ in both smooths will be

```
kernel("daniell", m = c(2, 2))
```

```
#> Daniell(2,2)
#> coef[-4] = 0.04
#> coef[-3] = 0.08
#> coef[-2] = 0.12
#> coef[-1] = 0.16
#> coef[ 0] = 0.20
```

```
#> coef[ 1] = 0.16
#> coef[ 2] = 0.12
#> coef[ 3] = 0.08
#> coef[ 4] = 0.04
```

```
kernel("modified.daniell", m = c(2, 2))
```

```
#> mDaniell(2,2)
#> coef[-4] = 0.0156
#> coef[-3] = 0.0625
#> coef[-2] = 0.1250
#> coef[-1] = 0.1875
#> coef[ 0] = 0.2188
#> coef[ 1] = 0.1875
#> coef[ 2] = 0.1250
#> coef[ 3] = 0.0625
#> coef[ 4] = 0.0156
```

The center values are weighted slightly more heavily in the modified Daniell kernel than in the unmodified one.

When we smooth a periodogram, we are smoothing *across frequencies* rather than across times.

Example: Spectral analysis of the airline passenger data

Recall the airline passenger time series (**?@fig-airpassangers**) that has an increasing trend and strong multiplicative seasonality.
The series should be detrended before a spectral analysis. For demonstration purposes, compute periodogram for the raw data and log-transformed and detrended.

```
# Fit a quadratic trend to log-transformed data
t <- as.vector(time(AirPassengers))
mod <- lm(log10(AirPassengers) ~ poly(t, degree = 2))
res <- ts(mod$residuals)
attributes(res) <- attributes(AirPassengers)

# Compute spectra
spec_raw <- spec.pgram(AirPassengers, detrend = FALSE, plot = FALSE)
spec_log_detrend <- spec.pgram(res, detrend = FALSE, plot = FALSE)
```

The x-axes of the periodograms correspond to $\omega/(2\pi)$ or the number of cycles per the time series period specified in the `ts` object (12 months). If the frequency of 12 was not specified for this time series, the x-axes would be different, and the first spectrum peak would correspond to ≈ 0.08.
Figure 19.1 B shows that the spectrum of the raw data is dominated by low frequencies (trend). After detrending the data (Figure 19.1 C), the spectrum at frequency 1 (meaning one cycle per year) is the dominant one.

```
pAirPassengers <- forecast::autoplot(AirPassengers, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers (thousand)") +
    ggtitle("Raw series")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
p3 <- forecast::autoplot(res, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers residuals (lg[thousand])") +
    ggtitle("Detrended and log-transformed series")
p4 <- data.frame(Spectrum = spec_log_detrend$spec,
                 Frequency = spec_log_detrend$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
(pAirPassengers + p2) / (p3 + p4) +
    plot_annotation(tag_levels = 'A')
```

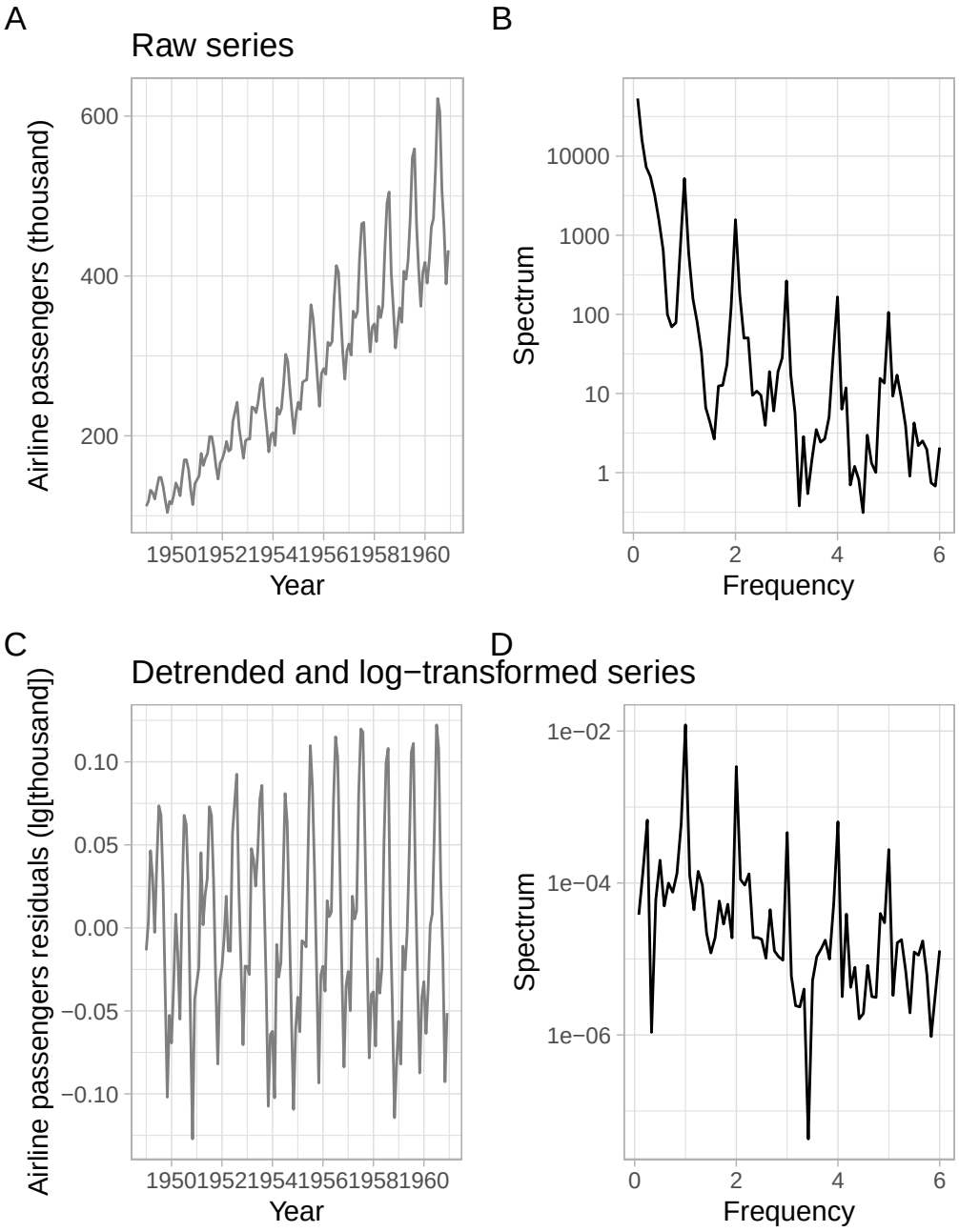


Figure 10.1.: Monthly `AirPassengers` time series, log-transformed and detrended, and the corresponding fast Fourier transforms.

Note that the function `spec.pgram()` has the option of removing a simpler (linear) trend and showing the results as a base-R plot. The confidence band in the upper right corner of this plot helps to identify the statistical significance of the peaks (Figure 19.2).

```
par(mfrow = c(2, 2))
spec.pgram(res, spans = c(3))
spec.pgram(res, spans = c(3, 3))
spec.pgram(res, spans = c(7, 7))
spec.pgram(res, spans = c(11, 11))
```

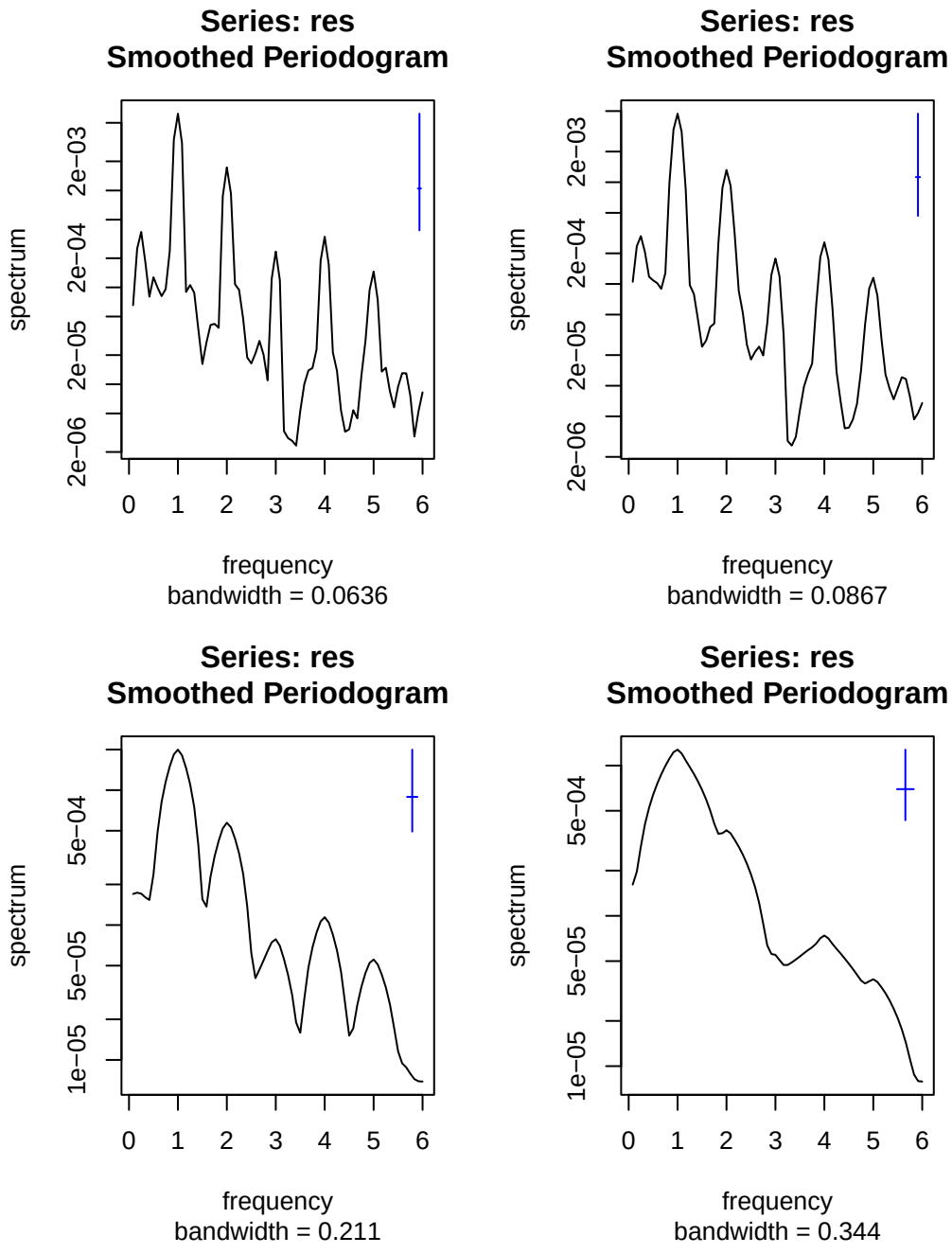


Figure 10.2.: Smoothed periodograms of monthly log-transformed and detrended `AirPassengers` time series.

Another method of testing the reality of a peak is to look at its harmonics. It is extremely unlikely that a true cycle will be shaped perfectly as a sine curve, hence at least a few of the first harmonics will show up as well. For example, in monthly data with annual seasonality (period of 12 months), we often observe peaks at 6, 4, and 3 months. This is exactly the pattern that we see in the figures above.
We can also approximate our data with an AR model and then plot the approximating periodogram of the AR model (Figure 19.3).

```
spectrum(res, method = "ar")
```

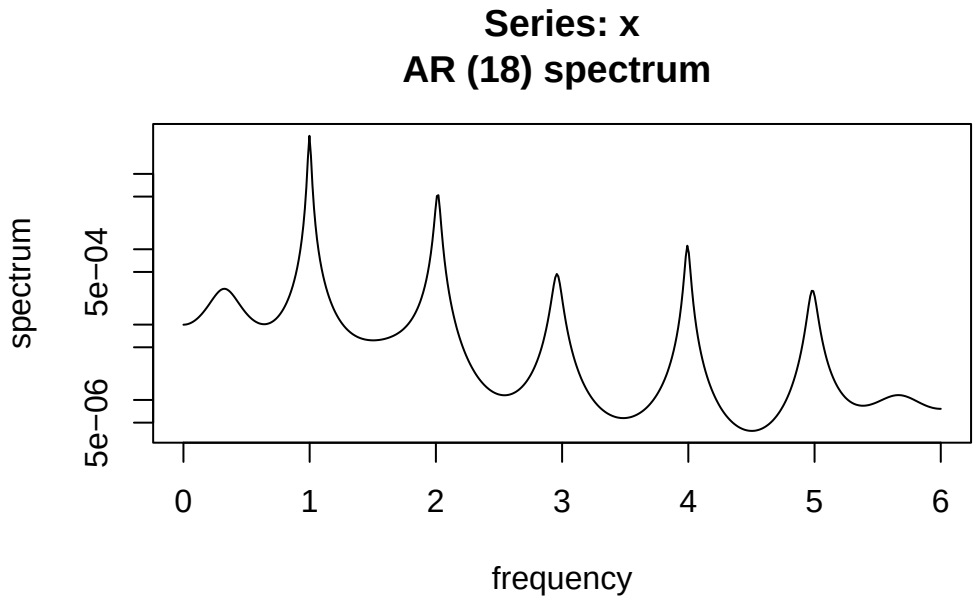


Figure 10.3.: Periodogram of AR process approximating the monthly log-transformed and detrended `AirPassengers` time series.

10.5. Periodogram for unequally-spaced time series

Unequally/unevenly-spaced time series or gaps in observations (missing data) can be handled with the Lomb–Scargle periodogram calculation. This method was originally developed for the analysis of unevenly-spaced astronomical signals (Lomb 1976; Scargle 1982) but found application in many other domains, including environmental science (e.g., Ruf 1999; Campos et al. 2008; Siskey et al. 2016).

Example: Ammonium concentration in wastewater system

For example, consider a time series `imputeTS::tsNH4` of ammonium (NH_4) concentration in a wastewater system (Figure 19.4). This time series contains observations from regular 10-minute

intervals, but some of the observations are missing.

```
forecast::autoplot(imputeTS::tsNH4) +
    xlab("Day") +
    ylab(bquote(NH[4]~concentration))
```

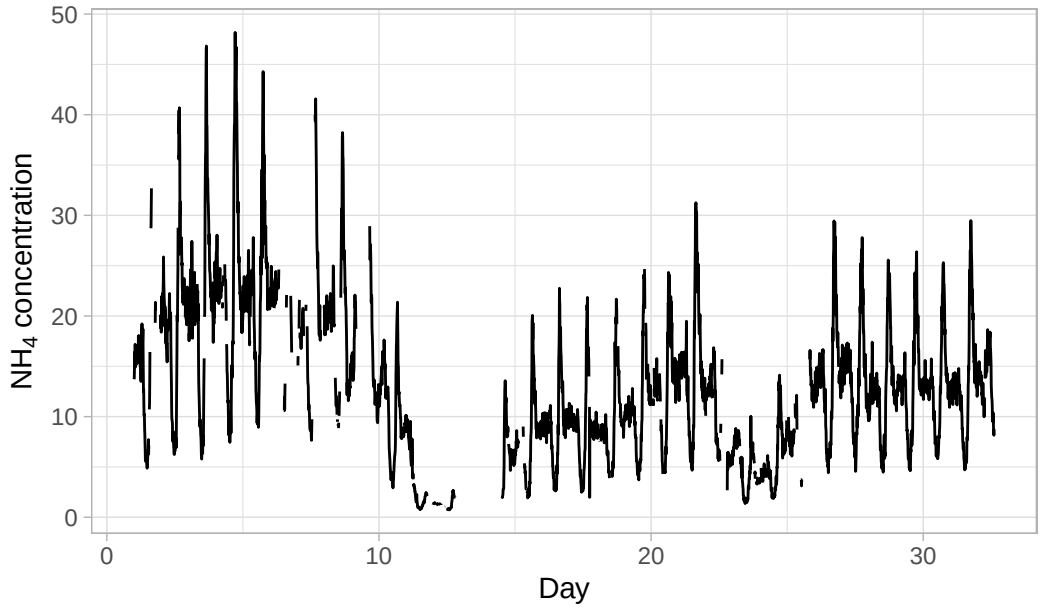


Figure 10.4.: Time series of ammonium concentration in a wastewater system.

Save the data in a format acceptable by the Lomb–Scargle function.

```
D <- data.frame(NH4 = as.vector(imputeTS::tsNH4),
                Time = as.vector(time(imputeTS::tsNH4)))
```

If needed, `na.omit(D)` can be used to remove the rows with missing values.
Figure 19.5 and Figure 19.6 show periodograms calculated based on these data. Note that the function `lomb::lsp()` has arguments adjusting the span of the frequencies and significance level.

```
par(mfrow = c(2, 2))
lomb::lsp(D$NH4, times = D$Time, type = "frequency", alpha = 0.05, main = "")
```

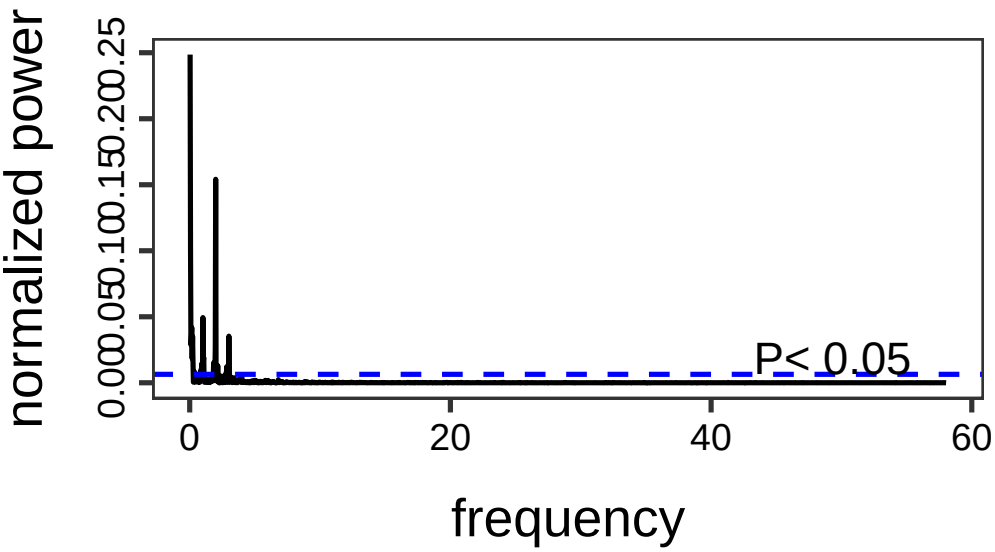


Figure 10.5.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

```
lomb::lsp(D$NH4, times = D$Time, type = "period", alpha = 0.05, main = "")
```

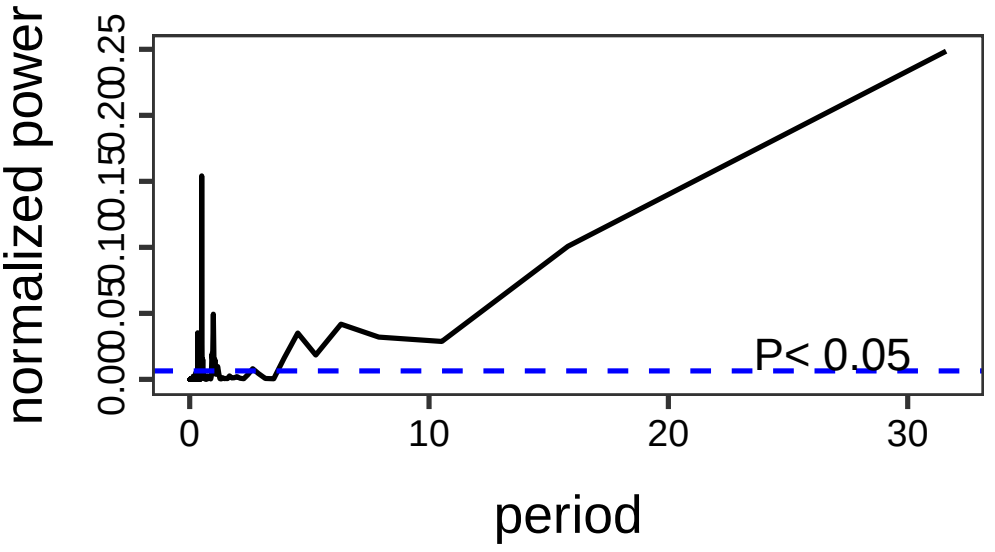


Figure 10.6.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

10.6. Frequency estimation in time

The assumption of a whole time series having a constant spectrum might be very limiting for the analysis of long time series. An estimation of frequencies locally in time allows us to study how the spectrum changes in time, and also detect smaller (short-lived) signals that would be averaged out otherwise. A straightforward solution is to separate the time series into windows (sub-periods) and estimate a spectrum in each such window.

The windowed analysis makes sense when in each window we have enough observations to estimate the frequencies. As a guide, we should have at least two observations per period to be able to represent the signal in the discrete Fourier transform. For example, we cannot estimate the seasonality if we sample once per year – this is our *sampling rate* or *sampling frequency* F_s. To start identifying seasonal periodicity, our sampling rate should be at least 2 samples per year. The highest frequency we can work with is limited by the *Nyquist frequency*

$$F_N = \frac{F_s}{2},$$

which is half of the sampling frequency.

After applying the FFT in each window, the results can be visualized in a *spectrogram*. The spectrogram combines information from multiple periodograms: it shows time on the x-axis, frequency on the y-axis, and the corresponding spectrum power as the color.

Example: Spectrograms for the NAO

Consider a long time series of the North Atlantic Oscillation (NAO) index obtained from an ensemble of 100 model-constrained NAO reconstructions (Figure 19.7).

```
NAOdf <- readr::read_csv("data/NAO.csv", skip = 12, show_col_types = FALSE) %>%
    rename(NAOproxy = nao_mc_mean) %>%
    select(Year, NAOproxy) %>%
    arrange(Year)
NAO <- ts(NAOdf$NAOproxy, start = NAOdf$Year[1])
spec_raw <- spec.pgram(NAO, detrend = FALSE, plot = FALSE, spans = 3)

# Find the frequency with the max power
fmax <- spec_raw$freq[which.max(spec_raw$spec)]
```

```
p1 <- forecast::autoplot(NAO) +
    xlab("Year") +
    ylab("NAO")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p3 <- data.frame(Spectrum = spec_raw$spec,
                 Period = 1/spec_raw$freq) %>%
    ggplot(aes(x = Period, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = 1/fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

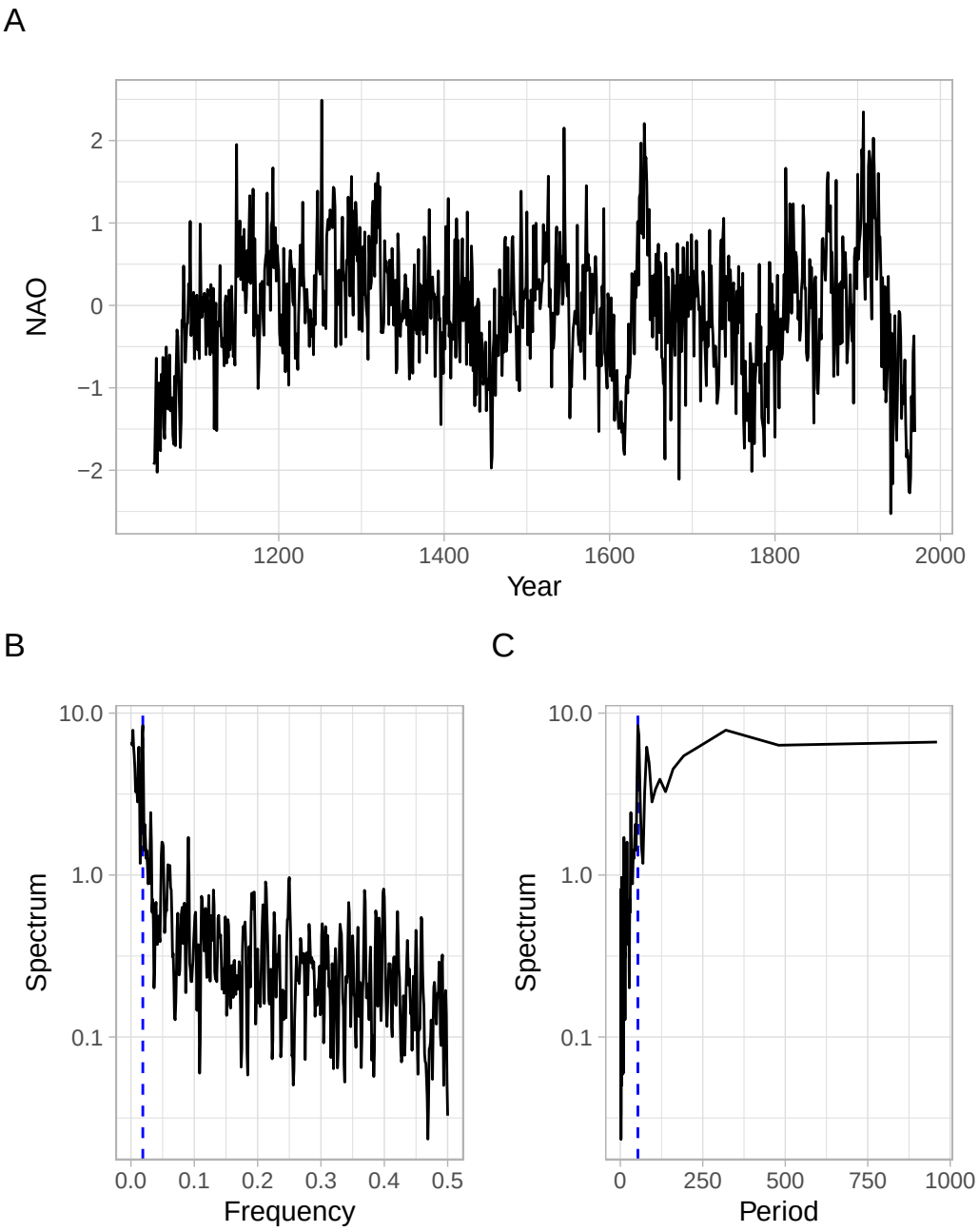


Figure 10.7.: North Atlantic Oscillation (NAO) winter index (December, January, and February) calculated as the mean of 100 model-constrained NAO reconstructions, along with its smoothed periodograms. The dashed lines denote the maximal magnitude of the spectrum.

This is an annual time series, hence the sampling frequency $F_s = 1$. From the global FFT results, the frequency with the maximal power is 0.019 year^{-1}, which is equivalent to the period of about 53.3 years. We will set the window for the FFT and calculate the spectrogram. Note

that we can use overlapping windows for a smoother picture.

```
# Set the window size (years), for applying the FFT in each window
window_size = 30

# Compute the spectrogram
SP <- signal::specgram(x = NAO,
                       n = window_size,
                       overlap = window_size/3,
                       Fs = 1)
```

See the spectrograms in Figure 19.8 and compare them with the time series plot in Figure 19.7 A.

```
par(mfrow = c(1, 2))

# Plot the spectrogram without decorations
print(SP, main = "A) Raw results")

# Discard phase information
P <- abs(SP$S)

# Normalize the spectrum to the range [0, 1]
P <- P - min(P)
P <- P/max(P)

# Extract time
Years <- time(NAO)[SP$t]

# Plot the spectrogram after processing
oce::imagep(x = Years,
            y = SP$f,
            z = t(P),
            col = oce::oce.colorsViridis,
            ylab = expression(Frequency~(y^-1 )),
            xlab = "Year",
            main = "B) After normalization and adding colors",
            drawPalette = TRUE,
            decimate = FALSE)
```

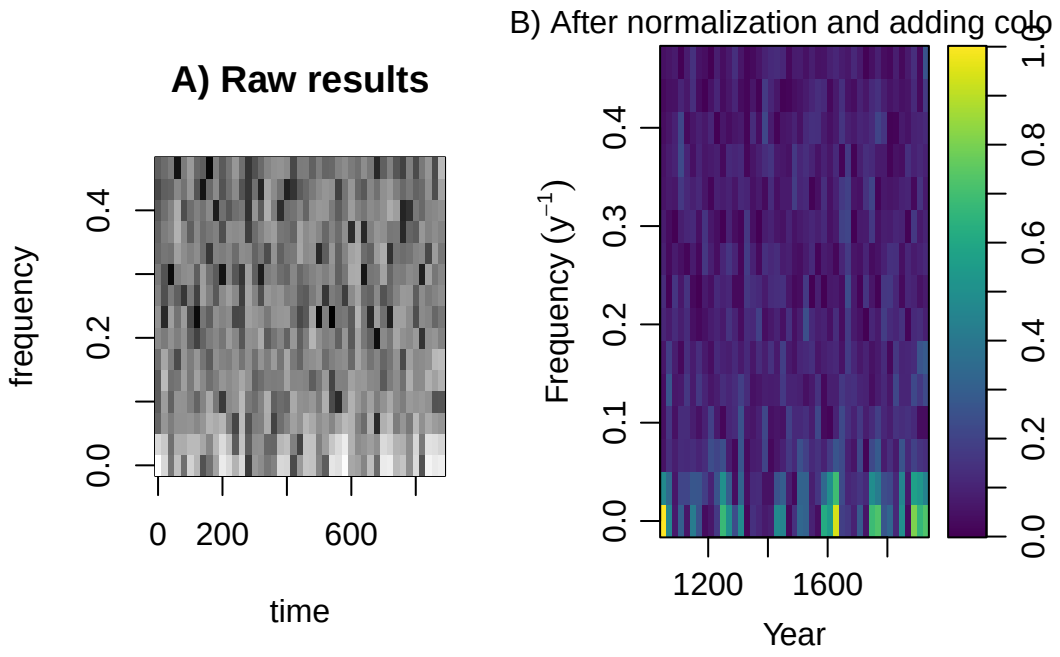


Figure 10.8.: Spectrograms for North Atlantic Oscillation (NAO) winter index.

10.7. Conclusion

For challenging problems, smoothing, multitapering, linear filtering, (repeated) pre-whitening, and Lomb–Scargle can be used together. Beware that aperiodic but autoregressive processes produce peaks in spectral densities. Harmonic analysis is a complicated art rather than a straightforward procedure.

It is extremely difficult to derive the significance of a weak periodicity from harmonic analysis. Do not believe analytical estimates (e.g., exponential probability), as they rarely apply to real data. It is essential to make simulations, typically permuting or bootstrapping the data keeping the observing times fixed. Simulations of the final model with the observation times are also advised.

10.8. Appendix

Wavelet analysis

An alternative to the windowed FFT is the wavelet analysis, which also estimates the strength of time series variations for different frequencies and times. *Wavelet* is a 'small wave' or function $\psi(t)$ approximating the variations. Popular wavelet functions are Meyer, Morlet, and Mexican hat.

Figure 19.9 shows the results of using the Morlet wavelet to process the NAO time series.

```
library(dplR)
wv <- morlet(y1 = NAOdf$NAOproxy, x1 = NAOdf$Year)
levs <- quantile(wv$Power, probs = c(0, 0.5, 0.75, 0.9, 0.95, 1))
wavelet.plot(wv, wavelet.levels = levs)
```

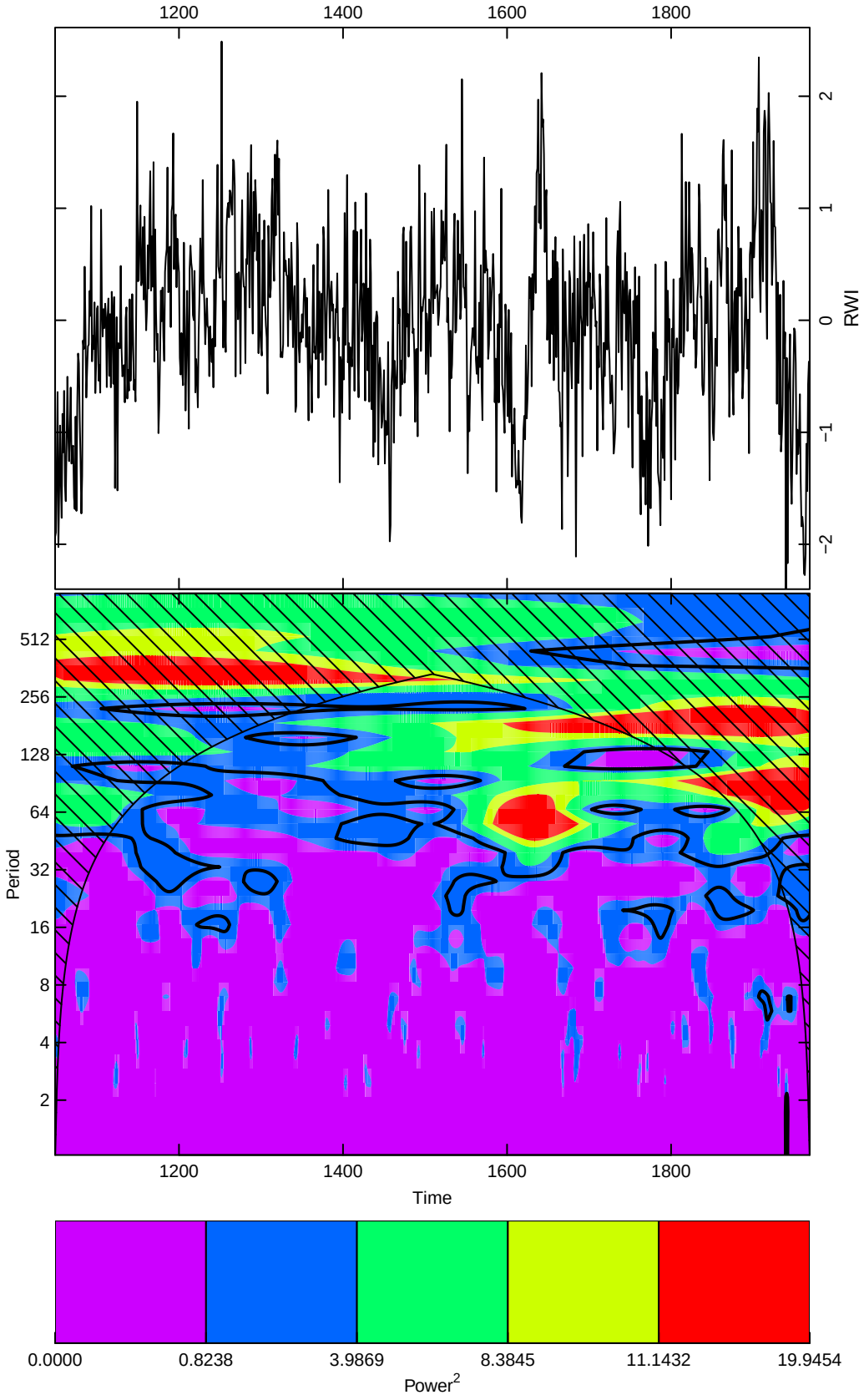


Figure 10.9.: Wavelet analysis of the North Atlantic Oscillation (NAO) winter index.

Part V.

Module 5 - Deep Reinforcement Learning

11. Deep Reinforcement Learning

The goal of this lecture is to introduce methods of handling nonstationary time series in regression models. You will become familiar with the problem of spurious correlation (regression) and approaches helping to avoid it.

Objectives

1. Learn three alternative ways of handling trends and/or seasonality to avoid spurious results: incorporate time effects into a regression model, use deviations from trends, or differenced series.
2. Introduce the concept of cointegration, learn how to detect it, and model using an error correction model.

Reading materials

- Chapter 10 in Wooldridge (2013)
- Chapter 6 in Kirchgässner and Wolters (2007) on cointegration

11.1. Spurious correlation

Results of statistical analysis (correlation, regression, etc.) are called *spurious* when they are likely driven not by the underlying mechanisms such as the physical relationships between variables, but by matching patterns (often, temporal patterns) of the variables leading to the statistical significance of tested relationships. Such matching patterns include trends, periodic fluctuations (seasonality), and more random patterns like spikes matching in several time series, for example, detected by searching over a large database of time series (a.k.a. cherry picking).

```
set.seed(123)
T <- 300
Xt <- ts(rnorm(T))
Yt <- ts(rnorm(T))
p1 <- forecast::autoplot(Xt)
p2 <- forecast::autoplot(Yt)
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

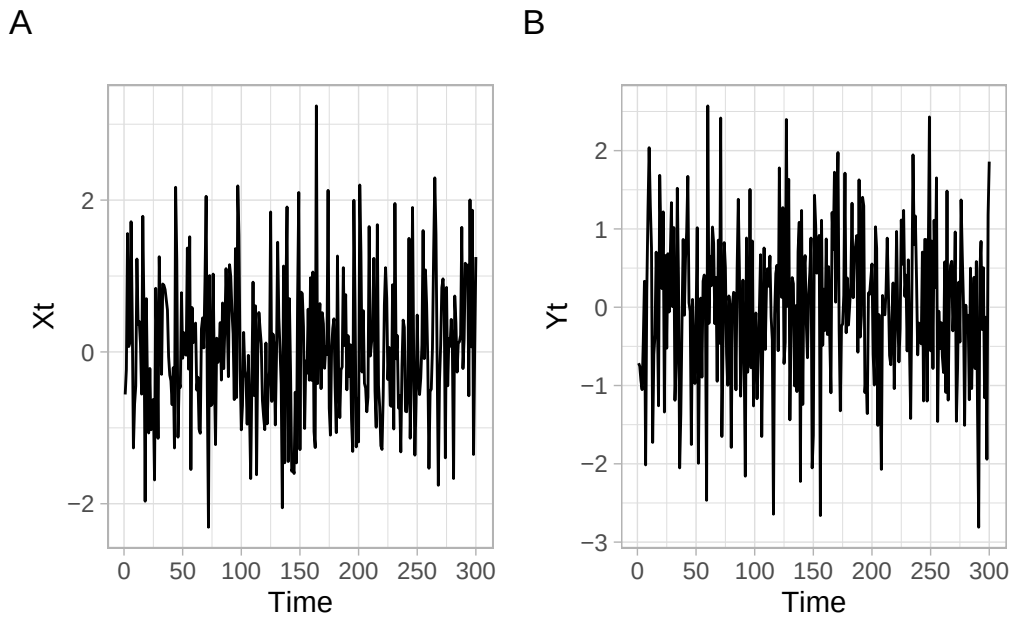


Figure 11.1.: Original stationary and independent time series.

Figure 11.1 shows two series of length $T = 300$ simulated from the standard normal distribution, $N(0, 1)$. These are independent and identically distributed (i.i.d.) random variables: each of the T values in X_t and Y_t was drawn independently from other values from the same distribution. Two random variables are independent if the realization of one does not affect the probability distribution of the other. Independence is a strong condition, it also implies (includes) that the values are not correlated. This is true both for the values within the series X_t and Y_t, and across X_t and Y_t (i.e., X_t and Y_t are not autocorrelated, nor correlated with each other). This is the asymptotic property of such a time series (as the sample size increases infinitely).

In finite samples, we may observe that a point estimate of the correlation coefficient even for such an ideal series is not exactly zero, but it will be usually not statistically significant. See the results (confidence interval and p-value) from the correlation t-test below:

```
cor.test(Xt, Yt)
```

```
#>
#>  Pearson's product-moment correlation
#>
#> data:  Xt and Yt
#> t = -1, df = 298, p-value = 0.3
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  -0.1727  0.0529
#> sample estimates:
#>     cor
#> -0.0607
```

Not many time series behave like that in real life – often we observe some trends. Let's add linear trends to our simulated data, for example, trends going in the same direction but with slopes of different magnitudes. Here we use linear increasing trends, i.e., with positive slopes Figure 11.2.

> **Note**
>
> It is probably a good idea to refrain from writing 'positive trends' (or 'negative trends') because they can be confused with 'good' or 'beneficial' trends. For example, a decrease in the unemployment rate is a positive (good) trend for a country, but it is a trend with a negative slope. Contrary, a linear trend in pollutant concentrations with a positive slope (going upward) shows a negative (worsening) tendency for the ecosystem.

```
Xt <- Xt + c(1:T)/95
Yt <- Yt + c(1:T)/50
p1 <- forecast::autoplot(Xt) +
    ylim(-2, 8)
p2 <- forecast::autoplot(Yt) +
    ylim(-2, 8)
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

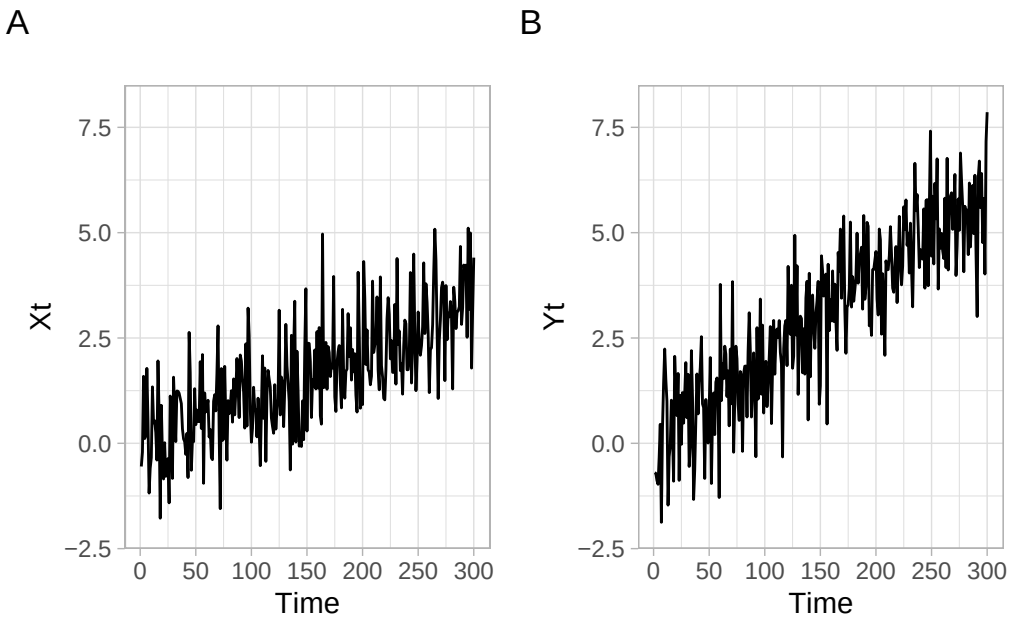


Figure 11.2.: Time series with trends.

After adding trends to each of the time series, the correlation of X_t and Y_t is strong and statistically significant, but this is not necessarily because these series are so strongly related to each other.

```
cor.test(Xt, Yt)
```

```
#>
#>  Pearson's product-moment correlation
#>
#> data:  Xt and Yt
#> t = 13, df = 298, p-value <2e-16
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  0.519 0.665
#> sample estimates:
#>   cor
#> 0.597
```

Some other factors may be driving the dynamics (trends) of X_t and Y_t in the same direction. Also, recall the general formula for computing correlation (or autocorrelation) of time series and that we need to subtract time-dependent means (not just a mean calculated over the whole period assuming the mean is not changing or time-independent), for example, as in **?@eq-Cov**.

For example, a city's growing population may result in more police officers and heavier pollution at the same time, partially because more people will drive their vehicles in the city. An attempt to correlate (regress) pollution with the number of police officers will produce a so-called spurious correlation (regression). The results of statistical tests will be likely statistically significant. However, is pollution directly related to the number of police officers? Will the dismissal of a police officer help to make the air cleaner?

Not only common increasing/decreasing trends but also other systematic changes (such as seasonality) may be responsible for spurious correlation effects. For example, both high ice cream sales and harmful algal blooms typically occur in warm weather conditions and may be 'significantly' correlated, suggesting banning ice cream for the sake of a safer environment. See more interesting examples of spurious correlation at http://www.tylervigen.com/spurious-correlations.

Sometimes, some simple tricks may help to avoid spurious results. For example, analyze not the raw numbers, but the *rates* that remove the effect of population growth in a city: crime rate per 100,000 inhabitants, number of people older than 70 per 1,000 inhabitants, etc. For more general approaches, see the next section.

11.2. Common approaches to regressing time series with trends

Consider a situation when we are given time series with trends (like in Figure 11.2), we do not know the data generating process (DGP; i.e., the true dependence structure), and we want to use these series in regression analysis.

In general, there are three alternative ways of dealing with trends in regression:

1. Incorporate time effect into the model;
2. Use deviations from trends (i.e., model and remove trends), or

3. Use differenced series (i.e., remove trends by differencing).

After these three approaches, here we consider a special case of cointegration (Section 11.3).

11.2.1. Incorporate time effects

Based on the time series plots (Figure 11.2), a linear time trend would fit these data, since we see a linear increase of values with time (so we add a linear time effect t in our model). Alternatively, e.g., if we observed parabolic structure, we could include $t + t^2$ or another form of trend.

```
t <- c(1:T)
mod_time <- lm(Yt ~ Xt + t)
summary(mod_time)
```

```
#>
#> Call:
#> lm(formula = Yt ~ Xt + t)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.7564 -0.6069  0.0546  0.6404  2.5802
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.000813   0.114665    0.01     0.99
#> Xt          -0.063741   0.060617   -1.05     0.29
#> t            0.020741   0.000939   22.09   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.99 on 297 degrees of freedom
#> Multiple R-squared:  0.756,  Adjusted R-squared:  0.755
#> F-statistic:  461 on 2 and 297 DF,  p-value: <2e-16
```

This model looks like this:

$$Y_t = \beta_0 + \beta_1 X_t + \beta_2 t + \epsilon_t,$$

estimated as:

$$\widehat{Y_t} = \hat{\beta}_0 + \hat{\beta}_1 X_t + \hat{\beta}_2 t, \tag{11.1}$$

$$\widehat{Y_t} = 0.0008 - 0.0637 X_t + 0.0207 t. \tag{11.2}$$

In the above model, the (highly statistically significant) time term took over the trend influence, thus, the coefficient for X shows the 'real' relationship between Y and X. Notice, the coefficient β_1 is not statistically significant, what we expected.

11.2.2. Use deviations from trends

Here we fit a separate time trend (may be of a different form for each time series: linear, quadratic, log, etc.) for each variable and find deviations from these trends. Based on Figure 11.2, linear trends are appropriate here:

$$Y_t = a_0 + a_1 t + e_{(Y)t}; \quad X_t = b_0 + b_1 t + e_{(X)t},$$

where $e_{(Y)t}$ and $e_{(X)t}$ are the trend residuals for the series Y_t and X_t, respectively.

After the trend coefficients a_0, a_1, b_0, and b_1 are estimated,

```
MY <- lm(Yt ~ t)
MX <- lm(Xt ~ t)
```

the smoothed series are

$$\widetilde{Y}_t = \hat{a}_0 + \hat{a}_1 t; \quad \widetilde{X}_t = \hat{b}_0 + \hat{b}_1 t$$

and the estimated trend residuals are (Figure 11.3)

$$\hat{e}_{(Y)t} = Y_t - \widetilde{Y}_t \quad \text{and} \quad \hat{e}_{(X)t} = X_t - \widetilde{X}_t.$$

```
p1 <- ggplot2::autoplot(as.ts(MY$resid)) +
    xlab("t") +
    ylab("eYt")
p2 <- ggplot2::autoplot(as.ts(MX$resid)) +
    xlab("t") +
    ylab("eXt")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

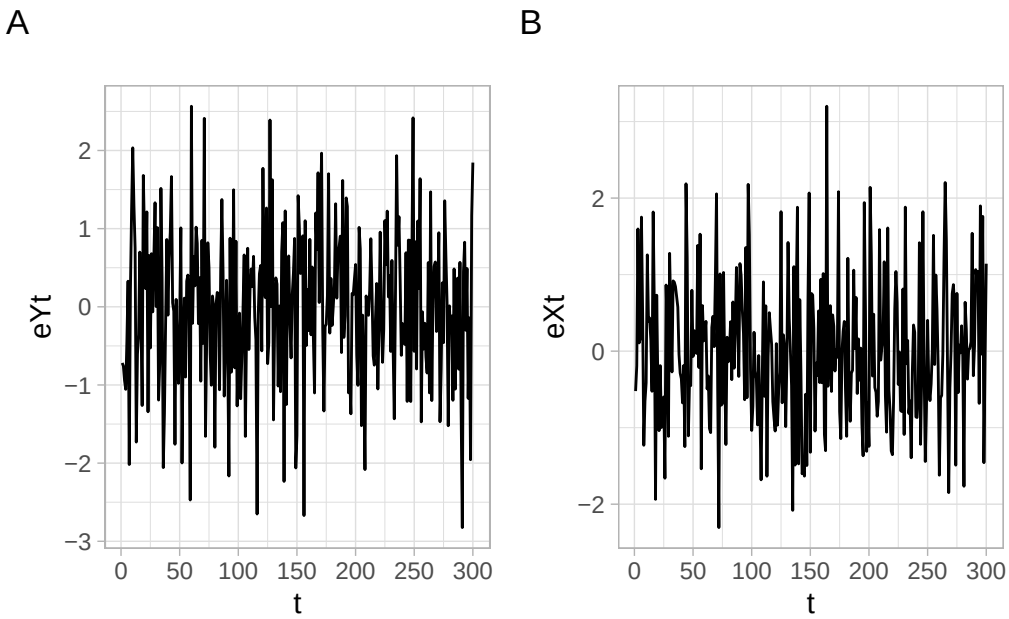


Figure 11.3.: Residuals from individually estimated linear trends.

Use the residuals in our regression model in place of the original variables:

$$\hat{e}_{(Y)t} = \beta_0 + \beta_1 \hat{e}_{(X)t} + \epsilon_t.$$

```
mod_devTrend <- lm(MY$residuals ~ MX$residuals)
summary(mod_devTrend)
```

```
#>
#> Call:
#> lm(formula = MY$residuals ~ MX$residuals)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -2.7564 -0.6069  0.0546  0.6404  2.5802
#>
#> Coefficients:
#>               Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   1.05e-17   5.71e-02    0.00     1.00
#> MX$residuals -6.37e-02   6.05e-02   -1.05     0.29
#>
#> Residual standard error: 0.989 on 298 degrees of freedom
#> Multiple R-squared:  0.00371,    Adjusted R-squared:  0.000366
#> F-statistic: 1.11 on 1 and 298 DF,  p-value: 0.293
```

We got the result (the relationship between the variables is not statistically significant) similar to running the model incorporating time trends in Section 11.2.1. Again, it is a reasonable result based on how the time series were simulated (independent, although with trends in the same direction).

11.2.3. Use differenced series

Instead of assuming a deterministic trend as in the previous subsections, we can try to eliminate a stochastic trend by differencing the time series. We define the lag-1 difference operator Δ by

$$\Delta X_t = X_t - X_{t-1} = (1 - B)X_t,$$

where B is the backward shift operator, $BX_t = X_{t-1}$.

There are tests developed in econometrics to find the appropriate order of differences (unit-root tests). Here, however, we will use the rule of thumb: for time trends looking linear (our case, see Figure 11.2) use the first-order differences, for parabolic shapes – the second-order differences. After differencing, the series should look stationary.

The first-order differences for our series (Figure 11.4) can be calculated as follows:

```
D1X <- diff(Xt)
D1Y <- diff(Yt)
```

```
p1 <- ggplot2::autoplot(D1Y) +
    xlab("t")
p2 <- ggplot2::autoplot(D1X) +
    xlab("t")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

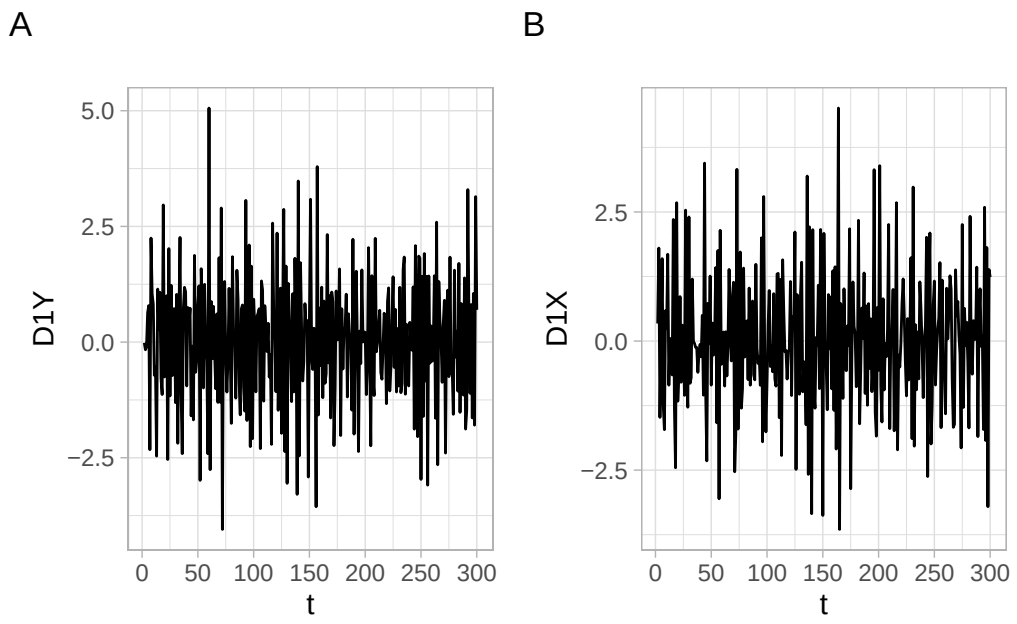


Figure 11.4.: First-order differences of the time series.

The series of first-order differences look stationary (Figure 11.4). Use these differenced series instead of the original time series in a regression model:

$$\Delta Y_t = \beta_0 + \beta_1 \Delta X_t + \epsilon_t.$$

```
mod_diff <- lm(D1Y ~ D1X)
summary(mod_diff)
```

```
#>
#> Call:
#> lm(formula = D1Y ~ D1X)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -4.056 -1.050   0.178   1.005   5.028
#>
#> Coefficients:
```

```
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   0.0284     0.0840    0.34     0.74
#> D1X           0.0114     0.0620    0.18     0.85
#>
#> Residual standard error: 1.45 on 297 degrees of freedom
#> Multiple R-squared:  0.000114,   Adjusted R-squared:  -0.00325
#> F-statistic: 0.0338 on 1 and 297 DF,  p-value: 0.854
```

As expected (since the original series X_t and Y_t, before adding the trend, were uncorrelated), the coefficient β_1 and the whole regression model are not statistically significant.

11.2.4. Wrong approach (do not repeat at home) leading to spurious regression

What if we forget about the three approaches above and just use the time series with trends in a regression model? This could be such a model:

$$Y_t = \beta_0 + \beta_1 X_t + \epsilon_t.$$

```
badModel <- lm(Yt ~ Xt)
summary(badModel)
```

```
#>
#> Call:
#> lm(formula = Yt ~ Xt)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -4.022 -1.020  0.098  1.219  4.091
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   1.5812     0.1454    10.9   <2e-16 ***
#> Xt            0.8883     0.0692    12.8   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 1.61 on 298 degrees of freedom
#> Multiple R-squared:  0.356,  Adjusted R-squared:  0.354
#> F-statistic:  165 on 1 and 298 DF,  p-value: <2e-16
```

The bad model shows spurious statistically significant effects, which are not true.

Beware of trends!

Example: Predicting sales of home appliances

Recall the dishwasher example from the first lecture. Let's use the above methods for regressing the time series of dishwasher shipments ($DISH_t$) and residential investments (RES_t). First, look at the time series plots of the raw data (Figure 11.5).

```
D <- read.delim("data/dish.txt") %>%
    rename(Year = YEAR)
p1 <- ggplot(D, aes(x = Year, y = DISH)) +
    geom_line()
p2 <- ggplot(D, aes(x = Year, y = RES)) +
    geom_line()
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

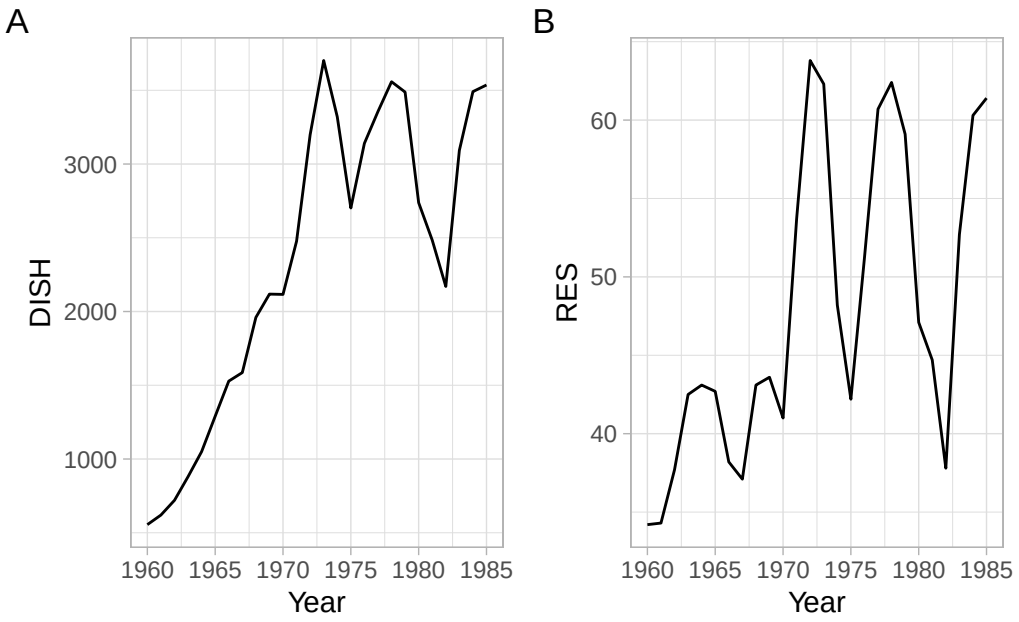


Figure 11.5.: Time series plots of the dishwasher shipments (DISH) and residential investments (RES).

Incorporate time effects

Incorporate linear trend using the model

$$DISH_t = \beta_0 + \beta_1 RES_t + \beta_2 Year + \epsilon_t; \tag{11.3}$$

see the residual diagnostics below and in Figure 11.6.

```
M_time <- lm(DISH ~ RES + Year, data = D)
summary(M_time)
```

```
#>
#> Call:
#> lm(formula = DISH ~ RES + Year, data = D)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -508.6 -237.7  -52.1  236.1  859.9
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -1.32e+05   2.32e+04   -5.71  8.1e-06 ***
#> RES          5.89e+01   9.35e+00    6.29  2.0e-06 ***
#> Year         6.69e+01   1.19e+01    5.63  1.0e-05 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 353 on 23 degrees of freedom
#> Multiple R-squared:  0.893,  Adjusted R-squared:  0.884
#> F-statistic: 96.3 on 2 and 23 DF,  p-value: 6.65e-12
```

```
shapiro.test(M_time$residuals)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  M_time$residuals
#> W = 1, p-value = 0.4
```

```
lawstat::runs.test(M_time$residuals, plot.it = FALSE)
```

```
#>
#>  Runs Test - Two sided
#>
#> data:  M_time$residuals
#> Standardized Runs Statistic = -4, p-value = 3e-04
```

```
p1 <- ggplot(D, aes(x = Year, y = M_time$residuals)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    ylab("Residuals")
p2 <- forecast::ggAcf(M_time$residuals) +
    ggtitle("") +
    xlab("Lag (years)")
p3 <- ggpubr::ggqqplot(M_time$residuals) +
    xlab("Standard normal quantiles")
p1 + p2 + p3 +
    plot_annotation(tag_levels = 'A')
```

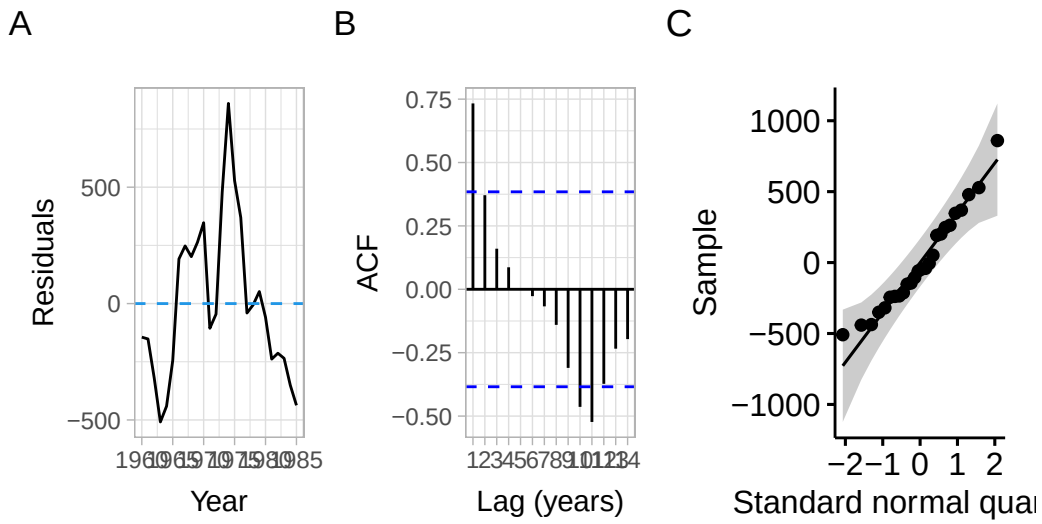


Figure 11.6.: Diagnostics plots for residuals from Equation 11.3.

Use differenced time series
Difference the time series (Figure 11.7) to use in the model

$$\Delta DISH_t = \beta_0 + \beta_1 \Delta RES_t + \epsilon_t \tag{11.4}$$

see the residual diagnostics below and in Figure 11.8.

```
D_DISH <- diff(D$DISH)
D_RES <- diff(D$RES)
M_diff <- lm(D_DISH ~ D_RES)
summary(M_diff)
```

```
#>
#> Call:
#> lm(formula = D_DISH ~ D_RES)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -424.4 -121.3    4.4   72.3  498.0
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)    71.22      43.16    1.65     0.11
#> D_RES          44.14       6.07    7.27  2.1e-07 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 213 on 23 degrees of freedom
```

```
#> Multiple R-squared:  0.697,  Adjusted R-squared:  0.684
#> F-statistic: 52.8 on 1 and 23 DF,  p-value: 2.13e-07
```

```
p1 <- ggplot(D[-1,], aes(x = Year, y = D_DISH)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4)
p2 <- ggplot(D[-1,], aes(x = Year, y = D_RES)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4)
p3 <- ggplot(data.frame(D_DISH, D_RES), aes(y = D_DISH, x = D_RES)) +
    geom_point()
p1 + p2 + p3 +
    plot_annotation(tag_levels = 'A')
```

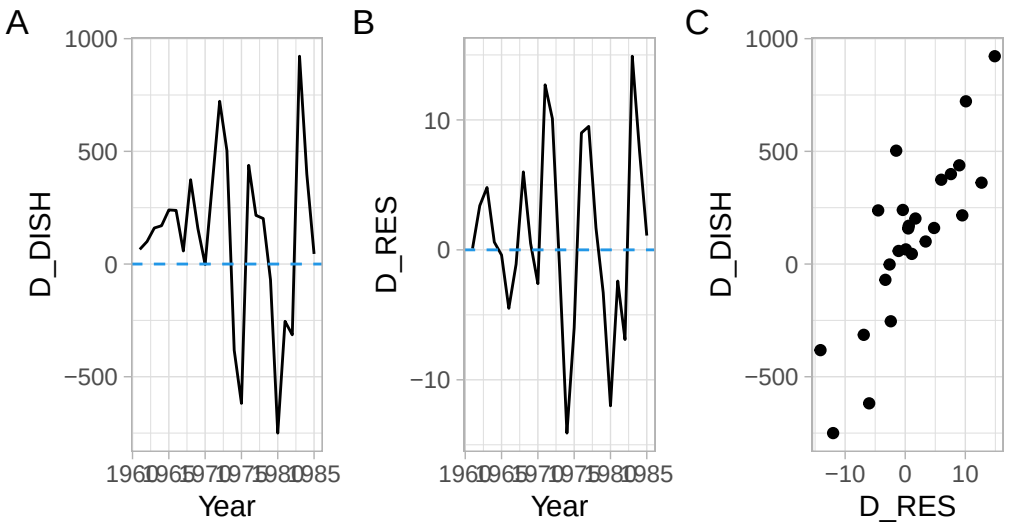


Figure 11.7.: Time differences of dishwasher shipments and residential investments, and a scatterplot for assessing pairwise relationships.

```
shapiro.test(M_diff$residuals)
```

```
#>
#>  Shapiro-Wilk normality test
#>
#> data:  M_diff$residuals
#> W = 1, p-value = 0.8
```

```
lawstat::runs.test(M_diff$residuals, plot.it = FALSE)
```

```
#>
```

```
#>  Runs Test - Two sided
#>
#> data:  M_diff$residuals
#> Standardized Runs Statistic = -2, p-value = 0.07
```

```
p1 <- ggplot(D[-1,], aes(x = Year, y = M_diff$residuals)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    ylab("Residuals")
p2 <- forecast::ggAcf(M_diff$residuals) +
    ggtitle("") +
    xlab("Lag (years)")
p3 <- ggpubr::ggqqplot(M_diff$residuals) +
    xlab("Standard normal quantiles")
p1 + p2 + p3 +
    plot_annotation(tag_levels = 'A')
```

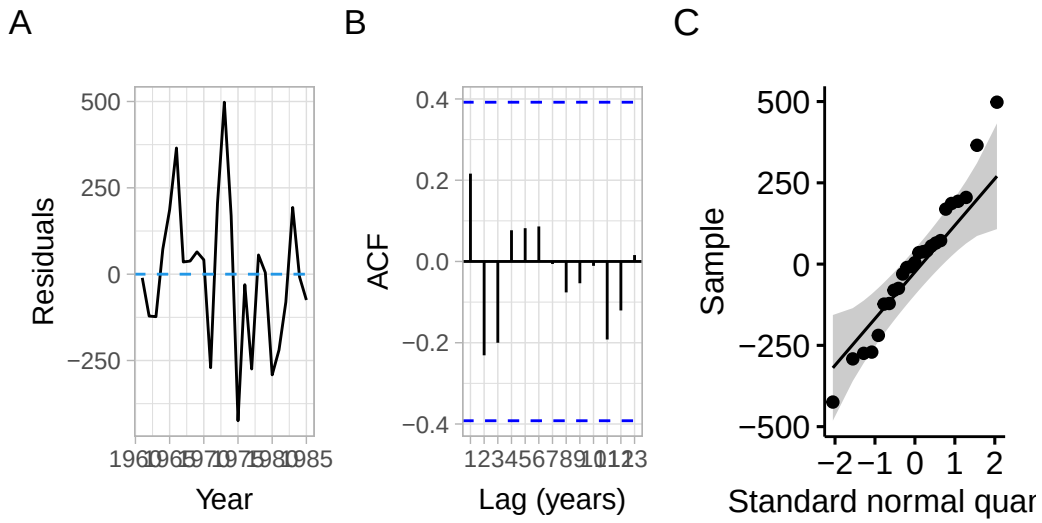


Figure 11.8.: Diagnostics plots for residuals from Equation 11.4.

11.3. Cointegration

Generally, cointegration might be characterized by two or more I(1) variables indicating a common long-run development, i.e., the variables do not drift away from each other except for transitory fluctuations. This defines a statistical equilibrium that, in empirical analysis, can often be interpreted as a long-run [economic] relation (Engle and Granger 1987).

In other words, two I(1) series X_t and Y_t are cointegrated if their linear combination u_t is I(0):

$$Y_t - \beta X_t = u_t. \tag{11.5}$$

Cointegration means a common stochastic trend (see Appendix C on testing for a common parametric trend). The vector $(1, -\beta)^\top$ is called the *cointegration vector*.

For the development of methods of analyzing time series cointegration, in 2003, Clive W. J. Granger was awarded 1/2 of the Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel (the other half was awarded to R. Engle, see **?@sec-GARCH**).

11.3.1. Two-step Engle–Granger method

1. Estimate long-run relationship, i.e., regression in levels as in Equation 11.5, and test residuals for I(0).
2. If the residual series u_t is I(0), use it in *error correction model* (ECM) regression

$$\begin{aligned} \Delta Y_t &= a_0 - \gamma_Y (Y_{t-1} - \beta X_{t-1}) + \sum_{j=1}^{n_X} a_{Xj} \Delta X_{t-j} + \sum_{j=1}^{n_Y} a_{Yj} \Delta Y_{t-j} + u_{Y,t}, \\ \Delta X_t &= b_0 + \gamma_X (Y_{t-1} - \beta X_{t-1}) + \sum_{j=1}^{k_X} b_{Xj} \Delta X_{t-j} + \sum_{j=1}^{k_Y} b_{Yj} \Delta Y_{t-j} + u_{X,t}, \end{aligned} \tag{11.6}$$

   where u_X and u_Y are pure random processes. If X_t and Y_t are cointegrated, at least one γ_i, $i = X, Y$, has to be different from zero.

OLS estimator is super consistent, convergence T. However, OLS can be biased in small samples.

The representation in Equation 11.6 has the advantage that it only contains stationary variables, although the underlying relation is between nonstationary (I(1)) variables. Thus, if the variables are cointegrated and the cointegration vector is known, the traditional statistical procedures can be applied for estimation and testing.

Example: Error correction model for simulated data

Demonstrate the analysis using simulated time series (Figure 11.9).

```
p1 <- ggplot2::autoplot(Xt)
p2 <- ggplot2::autoplot(Yt)
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

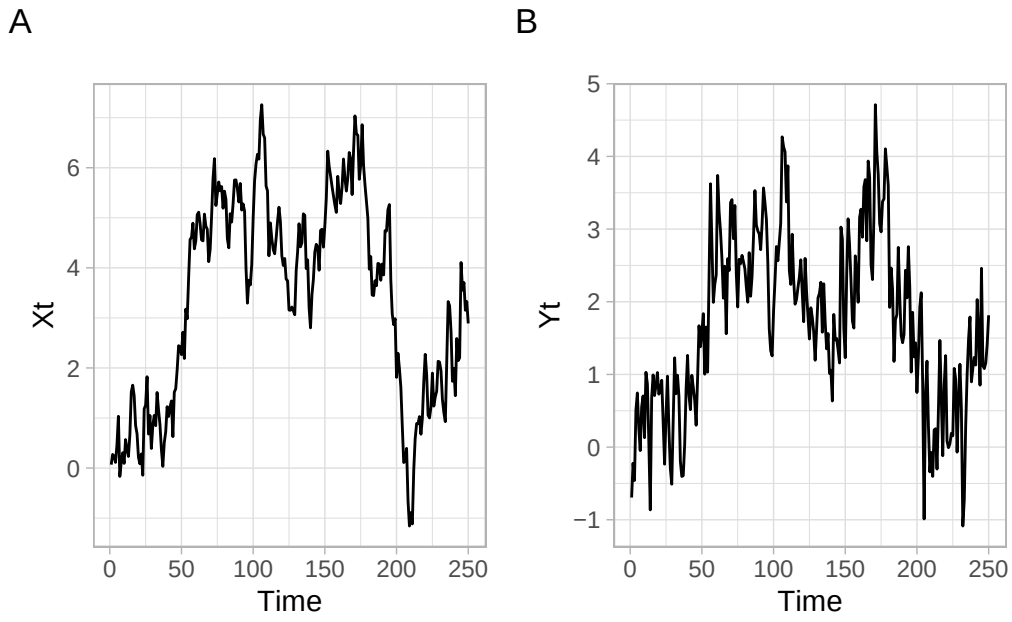


Figure 11.9.: Simulated I(1) time series.

Apply unit-root test to check the integration order of each series, using the R package `tseries`:

```
tseries::adf.test(Xt)
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  Xt
#> Dickey-Fuller = -2, Lag order = 6, p-value = 0.5
#> alternative hypothesis: stationary
```

```
tseries::adf.test(diff(Xt))
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  diff(Xt)
#> Dickey-Fuller = -7, Lag order = 6, p-value = 0.01
#> alternative hypothesis: stationary
```

```
tseries::adf.test(Yt)
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
```

```
#> data:  Yt
#> Dickey-Fuller = -2, Lag order = 6, p-value = 0.5
#> alternative hypothesis: stationary
```

```
tseries::adf.test(diff(Yt))
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  diff(Yt)
#> Dickey-Fuller = -8, Lag order = 6, p-value = 0.01
#> alternative hypothesis: stationary
```

With the confidence of 95%, the ADF test results show that each of the time series, X_t and Y_t, are I(1). (However, we have used the test 4 times, without controlling the overall Type I error.) Fit the linear regression

$$Y_t = a + bX_t + u_t. \tag{11.7}$$

The vector $[1, -b]$ is the cointegration vector.

```
Ut <- lm(Yt ~ Xt)$residuals
tseries::adf.test(Ut)
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  Ut
#> Dickey-Fuller = -5, Lag order = 6, p-value = 0.01
#> alternative hypothesis: stationary
```

While each of the time series X_t and Y_t is I(1), the resulting residual series $u_t \sim$ I(0), thus we conclude, X_t and Y_t are cointegrated.
Apply a simple error correction model (with $n_X = n_Y = 1$), using the R package `dynlm` or just specify lags using the package `dplyr`:

```
# Error correction term
ect <- Ut[-length(Ut)]

# Differenced series
dy <- diff(Yt)
dx <- diff(Xt)
```

Model using `dynlm::dynlm()`:

```
library(dynlm)
ecmdat1 <- cbind(dy, dx, ect)
ecm1 <- dynlm(dy ~ L(ect, 1) + L(dy, 1) + L(dx, 1), data = ecmdat1)
summary(ecm1)
```

```
#>
#> Time series regression with "ts" data:
#> Start = 3, End = 250
#>
#> Call:
#> dynlm(formula = dy ~ L(ect, 1) + L(dy, 1) + L(dx, 1), data = ecmdat1)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -1.8348 -0.3860 -0.0224  0.3696  1.5586
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  0.00919    0.03588    0.26   0.7980
#> L(ect, 1)   -0.67417    0.07482   -9.01   <2e-16 ***
#> L(dy, 1)    -0.49772    0.07341   -6.78    9e-11 ***
#> L(dx, 1)     0.22842    0.07906    2.89   0.0042 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.565 on 244 degrees of freedom
#> Multiple R-squared:  0.265,  Adjusted R-squared:  0.256
#> F-statistic: 29.3 on 3 and 244 DF,  p-value: 3.2e-16
```

Model using `lm()` and `dplyr::lag()`:

```
ecm2 <- lm(dy ~ dplyr::lag(ect, 1) +
               dplyr::lag(as.vector(dy), 1) +
               dplyr::lag(as.vector(dx), 1))
summary(ecm2)
```

```
#>
#> Call:
#> lm(formula = dy ~ dplyr::lag(ect, 1) + dplyr::lag(as.vector(dy),
#>     1) + dplyr::lag(as.vector(dx), 1))
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -1.8348 -0.3860 -0.0224  0.3696  1.5586
#>
#> Coefficients:
#>                                Estimate Std. Error t value Pr(>|t|)
#> (Intercept)                     0.00919    0.03588    0.26   0.7980
#> dplyr::lag(ect, 1)             -0.67417    0.07482   -9.01   <2e-16 ***
#> dplyr::lag(as.vector(dy), 1) -0.49772    0.07341   -6.78    9e-11 ***
#> dplyr::lag(as.vector(dx), 1)  0.22842    0.07906    2.89   0.0042 **
#> ---
```

```
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.565 on 244 degrees of freedom
#>   (1 observation deleted due to missingness)
#> Multiple R-squared:  0.265,  Adjusted R-squared:  0.256
#> F-statistic: 29.3 on 3 and 244 DF,  p-value: 3.2e-16
```

There is also the R package `ecm`, but it uses a modified formulation of the model, see details for the function `ecm::ecm()`.

In the example above, the time series were simulated as cointegrated. Below is an example of another I(1) process W_t but with a stochastic trend different from that of X_t. In this case, the linear combination of individually integrated W_t and X_t does not produce a stationary time series, thus, W_t and X_t are not cointegrated.

```
U2 <- lm(Wt ~ Xt)$residuals
tseries::adf.test(U2)
```

```
#>
#>  Augmented Dickey-Fuller Test
#>
#> data:  U2
#> Dickey-Fuller = -2, Lag order = 6, p-value = 0.7
#> alternative hypothesis: stationary
```

11.3.2. Johansen test

The Johansen test allows for more than one cointegrating relationship. The null hypothesis for the trace test is that the number of cointegration vectors is $r < k$, vs. the alternative that $r = k$. The testing proceeds sequentially for $k = 1, 2, ...$; and the first non-rejection of the null hypothesis is taken as an estimate of r.

Using the R package `urca`:

```
library(urca)
vecm <- ca.jo(cbind(Yt, Xt, Wt))
summary(vecm)
```

```
#>
#> ######################
#> # Johansen-Procedure #
#> ######################
#>
#> Test type: maximal eigenvalue statistic (lambda max) , with linear trend
#>
#> Eigenvalues (lambda):
```

```
#> [1] 0.270915 0.025774 0.000838
#>
#> Values of teststatistic and critical values of test:
#>
#>           test 10pct  5pct 1pct
#> r <= 2 |  0.21   6.5  8.18 11.7
#> r <= 1 |  6.48  12.9 14.90 19.2
#> r = 0  | 78.36  18.9 21.07 25.8
#>
#> Eigenvectors, normalised to first column:
#> (These are the cointegration relations)
#>
#>          Yt.l2  Xt.l2 Wt.l2
#> Yt.l2  1.00000  1.000   1.0
#> Xt.l2 -0.49209 -5.234 -26.2
#> Wt.l2 -0.00497  0.383 -19.3
#>
#> Weights W:
#> (This is the loading matrix)
#>
#>       Yt.l2   Xt.l2     Wt.l2
#> Yt.d -0.686 0.00228  2.64e-06
#> Xt.d -0.179 0.00796 -1.54e-05
#> Wt.d  0.180 0.00253  1.59e-04
```

If two time series are cointegrated, then the usual regression in Equation 11.7 is the so-called long-run equilibrium relation or attractor, i.e., the relationship between X_t and Y_t can be explained by Equation 11.7 in a long run. Equation 11.7 is applied for estimation, not for testing (see Figure 6.1 in Kirchgässner and Wolters 2007 on highly dispersed t-statistic). The error correction model in Equation 11.6 should be estimated for testing (p-values from the ECM can be used for testing, also see Chapter 6 in Kirchgässner and Wolters 2007).

11.4. Conclusion

Now we can incorporate trend effects into our models, using the three considered approaches or by testing for cointegration and applying an error correction model. The next step would be to incorporate autocorrelation structure in the residuals (the simulated example considered here used independent normally distributed noise, so it was an artificial ideal case of no autocorrelation, whereas we usually encounter autocorrelations, e.g., see residual diagnostics in the examples in Section 11.2.1).

Part VI.

Module 6 - Large Language Models

12. Large Language Models

13. Fine-Tuning and Adaptation of LLMs

Part VII.

Module 7 - Model Compression & Deployment

14. Model Compression, Deployment, and Efficiency

This lecture …

Objectives

1. Recognize the typical assumptions implied when making forecasts.
2. Distinguish *ex-post* from *ex-ante* forecasts.
3. Discuss peculiarities of assessing (cross-validating) regression models built for time series.
4. Define metrics used to assess quality of point forecasts and interval forecasts.
5. Design and implement cross-validation for time series models.

Reading materials

- Chapter 9 in Brockwell and Davis (2002)
- Chapter 7 in Hastie et al. (2009)

14.1. Time series forecasts

14.1.1. Assumptions

14.1.2. How to obtain forecasts from different types of models (white noise-like; recursive like ARIMA and Exponential smoothing; with x-variables and ARMA structure; Examples in R)

Univariate models - extrapolation of trends; ARIMA are 1-step-ahead forecasts (AIC)

Types of forecasts (point prediction or intervals; can be different intervals for the same point prediction)

14.1.3. Types of forecasts, especially when multiple regression

Types (*ex-post* from *ex-ante* forecasts)

Let $\hat{Y}_T(h)$ be a forecast h steps ahead made at the time T. If $\hat{Y}_T(h)$ only uses information up to time T, the resulting forecasts are called out-of-sample forecasts. Economists call them *ex-ante* forecasts. We have discussed several ways to select the optimal method or model for forecasting, e.g., using PMAE, PMSE, or coverage – all calculated on a testing set. Chatfield (2000) lists several ways to unfairly 'improve' forecasts:

1. Fitting the model to all the data including the test set.

2. Fitting several models to the training set and choosing the model which gives the best 'forecasts' of the test set. The selected model is then used (again) to produce forecasts of the test set, even though the latter has already been used in the modeling process.
3. Using the known test-set values of 'future' observations on the explanatory variables in multivariate forecasting. This will improve forecasts of the dependent variable in the test set, but these future values will not of course be known at the time the forecast is supposedly made (though in practice the 'forecast' is made at a later date). Economists call such forecasts *ex-post* forecasts to distinguish them from *ex-ante* forecasts. The latter, being genuinely out-of-sample, use forecasts of future values of explanatory variables, where necessary, to compute forecasts of the response variable. *Ex-post* forecasts can be useful for assessing the effects of explanatory variables, provided the analyst does not pretend that they are genuine out-of-sample forecasts.

So what to do if we put lots of effort to build a regression model using time series and need to forecast the response, Y_t, which is modeled using different independent variables $X_{t,k}$ $(k = 1, \dots, K)$? Two options are possible.

Leading indicators

If $X_{t,k}$'s are leading indicators with lags starting at l, we, generally, would not need their future values to obtain the forecasts $\hat{Y}_T(h)$, where $h \leqslant l$. For example, the model for losses tested in **?@sec-Granger** shows that precipitation with lag 1 is a good predictor for current losses, i.e., precipitation is a leading indicator. The 1-week ahead forecast of Y_{t+1} can be obtained using the current precipitation X_t (all data are available). If $h > l$, we will be forced to forecast the independent variables, $X_{t,k}$'s – see the next option.

Forecast of predictors

If we opt for forecasting $X_{t,k}$'s, the errors (uncertainty) of such forecasts will be larger, because future $X_{t,k}$'s themselves will be the estimates. Nevertheless, it might be the only choice when leading indicators are not available. Building a full and comprehensive model with all diagnostics for each regressor is usually unfeasible and even problematic if we plan to consider multivariate models for regressors (the complexity of models will quickly escalate). As an alternative, it is common to use automatic or semi-automatic univariate procedures that can help to forecast each of the $X_{t,k}$'s. For example, consider exponential smoothing, Holt–Winters smoothing, and auto-selected SARIMA/ARIMA/ARMA/AR/MA models – all those can be automated for a large number of forecasts to make.

14.2. Cross-validation schemes

Goals and intended implementation of the model hence the selection of the scheme.

Training - Testing (split %% and n)

Training - Testing - Evaluation

Window approach caret:: forecast::

14.3. Metrics for model comparison

How do we compare forecasting models to decide which one is better? We will look at various ways of choosing between models as the course progresses, but the most obvious answer is to see which one is better at predicting.

Suppose we have used the data $Y_1, \dots, Y_n$ to build M forecasting models $\hat{Y}_t^{(m)}$ $(m = 1, \dots, M)$ and we now obtain future observations $Y_{n+1}, \dots, Y_{n+k}$ that were not used to fit the models (also called *out-of-sample* data, after-sample, or the testing set; k is the size of this set). The difference $Y_t - \hat{Y}_t^{(m)}$ is the forecast (or prediction) error at the time t for the mth model. For each model, compute the prediction mean square error (PMSE)

$$PMSE_m = k^{-1} \sum_{t=n+1}^{n+k} \left(Y_t - \hat{Y}_t^{(m)}\right)^2 \tag{14.1}$$

and prediction mean absolute error (PMAE)

$$PMAE_m = k^{-1} \sum_{t=n+1}^{n+k} \left|Y_t - \hat{Y}_t^{(m)}\right| \tag{14.2}$$

and, similarly, prediction root mean square error (PRMSE; $PRMSE = \sqrt{PMSE}$), prediction mean absolute percentage error (PMAPE, if $Y_t \neq 0$ in the testing period), etc. We choose the model with the smallest error.

One obvious drawback to the above method is that it requires us to wait for future observations to compare models. A way around this is to take the historical dataset $Y_1, \dots, Y_n$ and split it into a *training set* $Y_1, \dots, Y_k$ and a *testing set* $Y_{k+1}, \dots, Y_n$, where $(n - k) \ll k$, i.e., most of the data goes into the training set.

> **Note**
>
> This scheme of splitting time series into the testing and training sets is a simple form of *cross-validation*. Not all forms of cross-validation apply to time series due to the usual temporal dependence in time series data. We need to select cross-validation techniques that can accommodate such dependence. Usually, it implies selecting data for validation not at random but in consecutive chunks (periods) and, ideally, with testing or validation periods being after the training period.

Forecasting models are then built using only the training set and used to 'forecast' values from the testing set. Sometimes it is called an *out-of-sample forecast* because we predict values for the times we have not used for the model specification and estimation. The testing set is used as a set of future observations to compute the PMSE. The PMSE and other errors are computed for each model over the testing set and then compared to see errors for which models are smaller.

If two models produce approximately the same errors, we choose the model that is simpler (involves fewer variables). This is called the *law of parsimony*.

The above error measures (PMSE, PRMSE, PMAE, etc.) compare observed and forecasted data points, hence, are measures of the accuracy of the *point forecasts*. Another way of comparing models could be based on the quality of their *interval forecasts*, i.e., by assessing how good the prediction

intervals are. To assess the quality of interval forecasts, one may start by computing the *empirical coverage* (proportion of observations in the testing set that are within – covered by – corresponding prediction intervals for given confidence C, e.g., 95%) and *average interval width*. Prediction intervals are well-calibrated if empirical coverage is close to C (more important) while intervals are not too wide (less important).

> **Note**
>
> To select the best coverage, one can calculate the absolute differences between the nominal coverage C and each empirical coverage $\hat{C}_m$:
>
> $$\Delta_m = |C - \hat{C}_m|.$$
>
> Hence, we select the model with the smallest Δ_m, not the largest coverage $\hat{C}_m$.

> **Note**
>
> It is possible to obtain different prediction intervals from the same model. For example, we can calculate prediction intervals based on normal and bootstrapped distributions. In this case, point forecasts are the same, but interval forecasts differ.

We may compare a great number of models using the training set, and choose the best one (with the smallest errors), however, it would be unfair to use the out-of-sample errors from the testing set for demonstrating the model performance because this part of the sample was used to select the model. Thus, it is advisable to have one more chunk of the same time series that was not used for model specification, estimation, or selection. Errors of the selected model on this *validation set* will be closer to the true (genuine) out-of-sample errors and can be used to improve coverage of true out-of-sample forecasts when the model is finally deployed.

14.4. Worked out example of comparing several models

1. Basic (naive) models: average / climatology / HWinters
2. Commonly or previously used model (GLM)
3. State-of-the-science or proposed model

14.5. Conclusion

15. Scalability & Optimization

The goal of this lecture is to learn how to view time series from the frequency perspective. You should be able to recognize features of time series from a periodogram similar to how we did it using plots of the autocorrelation function (ACF).

Objectives

1. Describe the mechanics of Fourier transform for time series.
2. Present results of spectrum calculations using a periodogram or spectrogram.
3. Give examples of smoothed periodograms.
4. Demonstrate the use of the Lomb–Scargle periodogram for analysis of unequally-spaced time series.

Reading materials

- Chapter 4 in Shumway and Stoffer (2017)
- Nason (2008) on wavelet analysis

15.1. Introduction

By now, we have been working in the *time domain*, such that our analysis could be seen as a regression of the present on the past (for example, ARIMA models). We have been using many time series plots with time on the x-axis.

An alternative approach is to analyze time series in a *spectral domain*, such that use a regression of present on a linear combination of sine and cosine functions. In this type of analysis, we often use periodogram plots, with frequency or period on the x-axis.

15.2. Regression on sinusoidal components

The simplest form of spectral analysis consists of regression on a periodic component:

$$Y_t = A\cos\omega t + B\sin\omega t + C + \epsilon_t, \tag{15.1}$$

where $t = 1, \dots, T$ and $\epsilon_t \sim \text{WN}(0, \sigma^2)$. Without loss of generality, we assume $0 \leqslant \omega \leqslant \pi$. In fact, for discrete data, frequencies outside this range are aliased into this range. For example, suppose that $-\pi < (\omega = \pi - \delta) < 0$, then

$$\begin{aligned}\cos(\omega t) &= \cos((\pi - \delta)t) \\ &= \cos(\pi t)\cos(\delta t) + \sin(\pi t)\sin(\delta t) \\ &= \cos(\delta t).\end{aligned}$$

Hence, a sampled sinusoid with a frequency smaller than 0 appears to coincide with a sinusoid with frequency in the interval $[0, \pi]$.

> **Note**
>
> The term $A\cos\omega t + B\sin\omega t$ is a periodic function with the period $2\pi/\omega$. The period $2\pi/\omega$ represents the number of time units that it takes for the function to take the same value again, i.e., to complete a cycle. The frequency, measured in cycles per time unit, is given by the inverse $\omega/(2\pi)$. The angular frequency, measured in radians per time unit, is given by ω. Because of its convenience, the angular frequency ω will be used to describe the periodicity of the function, and its name is shortened to frequency when there is no danger of confusion.

Consider monthly data that exhibit a 12-month seasonality. Hence, the period $2\pi/\omega = 12$, which implies the angular frequency $\omega = \pi/6$. The frequency, measured in cycles per time unit, is given by the inverse

$$\frac{\omega}{2\pi} = \frac{1}{12} \approx 0.08.$$

The formulas to estimate parameters of regression in Equation 19.1 take a much simpler form if ω is one of the Fourier frequencies, defined by

$$\omega_j = \frac{2\pi j}{T}, \quad j = 0, \dots, \frac{T}{2},$$

then

$$\begin{aligned}
\hat{A} &= \frac{2}{T}\sum_t Y_t \cos\omega_j t, \\
\hat{B} &= \frac{2}{T}\sum_t Y_t \sin\omega_j t, \\
\hat{C} &= \overline{Y} = \frac{1}{T}\sum_t Y_t.
\end{aligned}$$

A suitable way of testing the significance of the sinusoidal component with frequency ω_j is using its contribution to the sum of squares

$$R_T(\omega_j) = \frac{T}{2}\left(\hat{A}^2 + \hat{B}^2\right).$$

If the $\epsilon_t \sim N(0, \sigma^2)$, then it follows that $\hat{A}$ and $\hat{B}$ are also independent normal, each with the variance $2\sigma^2/T$, so under the null hypothesis of $A = B = 0$ we find that

$$\frac{R_T(\omega_j)}{\sigma^2} \sim \chi^2_2$$

or equivalently that $R_T(\omega_j)/(2\sigma^2)$ has an exponential distribution with mean 1. The above theory can be extended to the simultaneous estimation of several periodic components.

15.3. Periodogram

The *Fourier transform* uses Fourier series, such as the pairs of sines and cosines with different periods, to describe the frequencies present in the original time series.

The Fourier transform applied to an equally-spaced time series Y_t (where $t = 1, \dots, T$) is also called the *discrete Fourier transform* (DFT) because the time is discrete (not the values Y_t).

To reduce the computational complexity of DFT and speed up the computations, one of the *fast Fourier transform* (FFT) algorithms is typically used.

The results of the Fourier transform are shown in a periodogram that describes the spectral properties of the signal.

The periodogram is defined as

$$I_T(\omega) = \frac{1}{2\pi T} \left| \sum_{t=1}^{T} Y_t e^{i\omega t} \right|^2 ,$$

which is an approximately unbiased estimator of the spectral density f.

Some undesirable features of the periodogram:

1. $I_T(\omega)$ for fixed ω is not a consistent estimate of $f(\omega)$, since

$$I_T(\omega_j) \sim \frac{f(\omega_j)}{2} \chi_2^2.$$

   Therefore, the variance of $f^2(\omega)$ does not tend to 0 as $T \to \infty$.
2. The independence of periodogram ordinates at different Fourier frequencies suggests that the sample periodogram plotted as a function of ω will be extremely irregular.

Suppose that $\gamma(h)$ is the autocovariance function of a stationary process and that $f(\omega)$ is the spectral density for the same process (h is the time lag and ω is the frequency). The autocovariance $\gamma(h)$ and the spectral density $f(\omega)$ are related:

$$\gamma(h) = \int_{-1/2}^{1/2} e^{2\pi i \omega h} f(\omega) d\omega,$$

and

$$f(\omega) = \sum_{h=-\infty}^{+\infty} \gamma(h) e^{-2\pi i \omega h}.$$

In the language of advanced calculus, the autocovariance and spectral density are Fourier transform pairs. These Fourier transform equations show that there is a direct link between the time domain representation and the frequency domain representation of a time series.

15.4. Smoothing

The idea behind smoothing is to take weighted averages over neighboring frequencies to reduce the variability associated with individual periodogram values. However, such an operation necessarily introduces some bias into the estimation procedure. Theoretical studies focus on the amount of smoothing that is required to obtain an optimum trade-off between bias and variance. In practice, this usually means that the choice of a kernel and amount of smoothing is somewhat subjective.

The main form of a smoothed estimator is given by

$$\hat{f}(\lambda) = \int_{-\pi}^{\pi} \frac{1}{h} K\left(\frac{\omega - \lambda}{h}\right) I_T(\omega) d\omega,$$

where $I_T(\cdot)$ is the periodogram based on T observations, $K(\cdot)$ is a kernel function, and h is the bandwidth. We usually take $K(\cdot)$ to be a nonnegative function, symmetric about 0 and integrating to 1. Thus, any symmetric density, such as the normal density, will work. In practice, however, it is more usual to take a kernel of finite range, such as the *Epanechnikov kernel*

$$K(x) = \frac{3}{4\sqrt{5}}\left(1 - \frac{x^2}{5}\right), \quad -\sqrt{5} \leqslant x \leqslant \sqrt{5},$$

which is 0 outside the interval $[-\sqrt{5}, \sqrt{5}]$. This choice of kernel function has some optimality properties. However, in practice, this optimality is less important than the choice of bandwidth h, which effectively controls the range over which the periodogram is smoothed.

There are some additional difficulties with the performance of the sample periodogram in the presence of a sinusoidal variation whose frequency is not one of the Fourier frequencies. This effect is known as *leakage*. The reason of leakage is that we always consider a truncated periodogram. Truncation implicitly assumes that the time series is periodic with a period T, which, of course, is not always true. So we artificially introduce non-existent periodicities into the estimated spectrum, i.e., cause 'leakage' of the spectrum. (If the time series is perfectly periodic over T then there is no leakage.) The leakage can be treated using an operation of tapering on the periodogram, i.e., by choosing appropriate periodogram windows.

> **Note**
>
> When we work with periodograms, we lose all phase (relative location/time origin) information: the periodogram will be the same if all the data were circularly rotated to a new time origin, i.e., the observed data are treated as perfectly periodic.

Another popular choice to smooth a periodogram is the *Daniell kernel* (with parameter m). For a time series, it is a centered moving average of values between the times $t - m$ and $t + m$ (inclusive). For example, the smoothing formula for a Daniell kernel with $m = 2$ is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2}}{5} \\ &= 0.2x_{t-2} + 0.2x_{t-1} + 0.2x_t + 0.2x_{t+1} + 0.2x_{t+2}.\end{aligned}$$

The weighting coefficients for a Daniell kernel with $m = 2$ can be checked using the function `kernel()`. In the output, the indices in `coef[ ]` refer to the time difference from the center of the average at the time t.

```
kernel("daniell", m = 2)
```

```
#> Daniell(2)
#> coef[-2] = 0.2
#> coef[-1] = 0.2
#> coef[ 0] = 0.2
#> coef[ 1] = 0.2
#> coef[ 2] = 0.2
```

The modified Daniell kernel is such that the two endpoints in the averaging receive half the weight that the interior points do. For a modified Daniell kernel with $m = 2$, the smoothing is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + 2x_{t-1} + 2x_t + 2x_{t+1} + x_{t+2}}{8} \\ &= 0.125x_{t-2} + 0.25x_{t-1} + 0.25x_t + 0.25x_{t+1} + 0.125x_{t+2}\end{aligned}$$

List the weighting coefficients:

```
kernel("modified.daniell", m = 2)
```

```
#> mDaniell(2)
#> coef[-2] = 0.125
#> coef[-1] = 0.250
#> coef[ 0] = 0.250
#> coef[ 1] = 0.250
#> coef[ 2] = 0.125
```

Either the Daniell kernel or the modified Daniell kernel can be convoluted (repeated) so that the smoothing is applied again to the smoothed values. This produces a more extensive smoothing by averaging over a wider time interval. For instance, to repeat a Daniell kernel with $m = 2$ on the smoothed values that resulted from a Daniell kernel with $m = 2$, the formula would be

$$\hat{\hat{x}}_t = \frac{\hat{x}_{t-2} + \hat{x}_{t-1} + \hat{x}_t + \hat{x}_{t+1} + \hat{x}_{t+2}}{5}.$$

The weights for averaging the original data for convoluted Daniell kernels with $m = 2$ in both smooths will be

```
kernel("daniell", m = c(2, 2))
```

```
#> Daniell(2,2)
#> coef[-4] = 0.04
#> coef[-3] = 0.08
#> coef[-2] = 0.12
#> coef[-1] = 0.16
#> coef[ 0] = 0.20
```

```
#> coef[ 1] = 0.16
#> coef[ 2] = 0.12
#> coef[ 3] = 0.08
#> coef[ 4] = 0.04
```

```
kernel("modified.daniell", m = c(2, 2))
```

```
#> mDaniell(2,2)
#> coef[-4] = 0.0156
#> coef[-3] = 0.0625
#> coef[-2] = 0.1250
#> coef[-1] = 0.1875
#> coef[ 0] = 0.2188
#> coef[ 1] = 0.1875
#> coef[ 2] = 0.1250
#> coef[ 3] = 0.0625
#> coef[ 4] = 0.0156
```

The center values are weighted slightly more heavily in the modified Daniell kernel than in the unmodified one.

When we smooth a periodogram, we are smoothing *across frequencies* rather than across times.

Example: Spectral analysis of the airline passenger data

Recall the airline passenger time series (**?@fig-airpassangers**) that has an increasing trend and strong multiplicative seasonality.
The series should be detrended before a spectral analysis. For demonstration purposes, compute periodogram for the raw data and log-transformed and detrended.

```
# Fit a quadratic trend to log-transformed data
t <- as.vector(time(AirPassengers))
mod <- lm(log10(AirPassengers) ~ poly(t, degree = 2))
res <- ts(mod$residuals)
attributes(res) <- attributes(AirPassengers)

# Compute spectra
spec_raw <- spec.pgram(AirPassengers, detrend = FALSE, plot = FALSE)
spec_log_detrend <- spec.pgram(res, detrend = FALSE, plot = FALSE)
```

The x-axes of the periodograms correspond to $\omega/(2\pi)$ or the number of cycles per the time series period specified in the `ts` object (12 months). If the frequency of 12 was not specified for this time series, the x-axes would be different, and the first spectrum peak would correspond to ≈ 0.08.
Figure 19.1 B shows that the spectrum of the raw data is dominated by low frequencies (trend). After detrending the data (Figure 19.1 C), the spectrum at frequency 1 (meaning one cycle per year) is the dominant one.

```
pAirPassengers <- forecast::autoplot(AirPassengers, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers (thousand)") +
    ggtitle("Raw series")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
p3 <- forecast::autoplot(res, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers residuals (lg[thousand])") +
    ggtitle("Detrended and log-transformed series")
p4 <- data.frame(Spectrum = spec_log_detrend$spec,
                 Frequency = spec_log_detrend$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
(pAirPassengers + p2) / (p3 + p4) +
    plot_annotation(tag_levels = 'A')
```

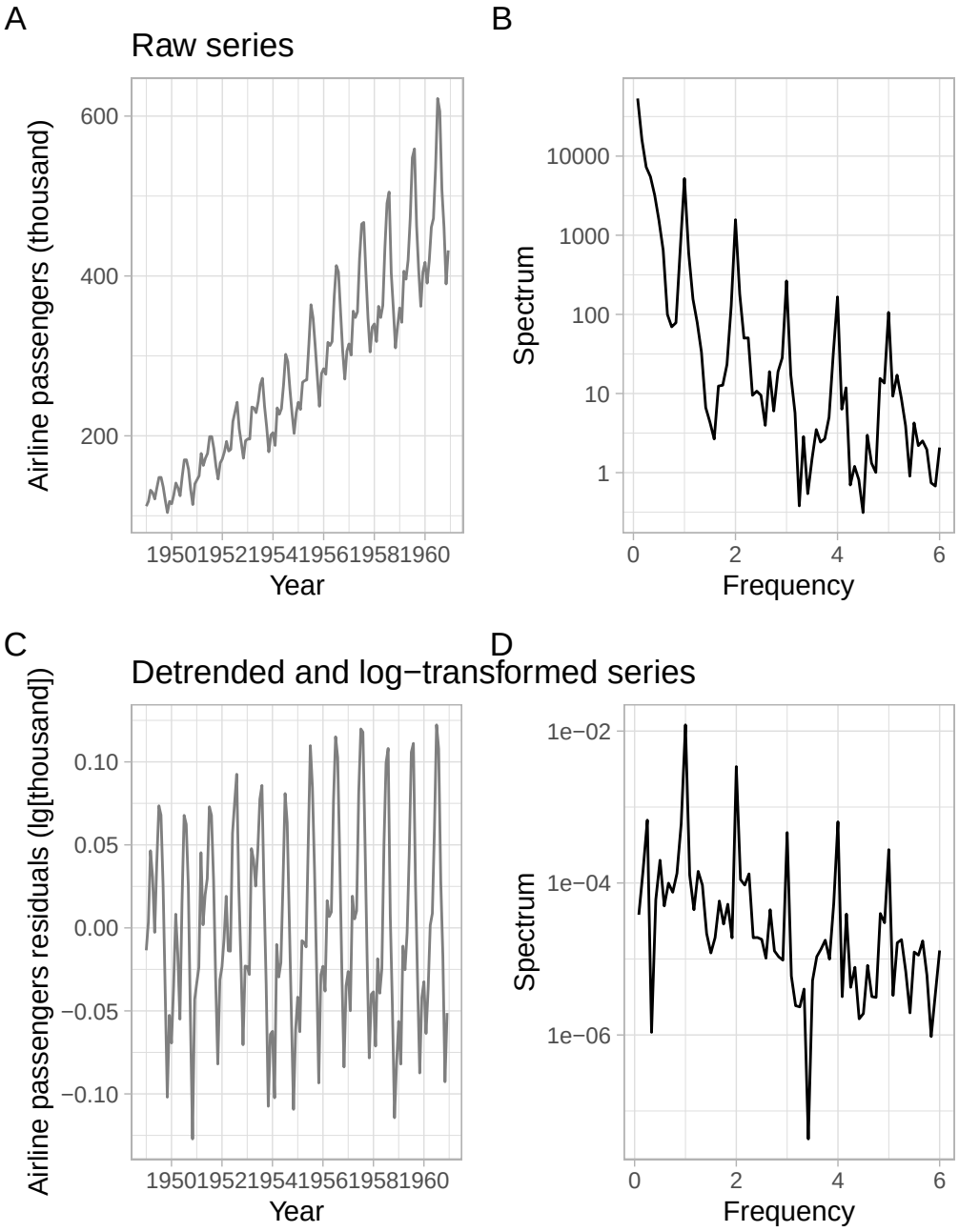


Figure 15.1.: Monthly `AirPassengers` time series, log-transformed and detrended, and the corresponding fast Fourier transforms.

Note that the function `spec.pgram()` has the option of removing a simpler (linear) trend and showing the results as a base-R plot. The confidence band in the upper right corner of this plot helps to identify the statistical significance of the peaks (Figure 19.2).

```r
par(mfrow = c(2, 2))
spec.pgram(res, spans = c(3))
spec.pgram(res, spans = c(3, 3))
spec.pgram(res, spans = c(7, 7))
spec.pgram(res, spans = c(11, 11))
```

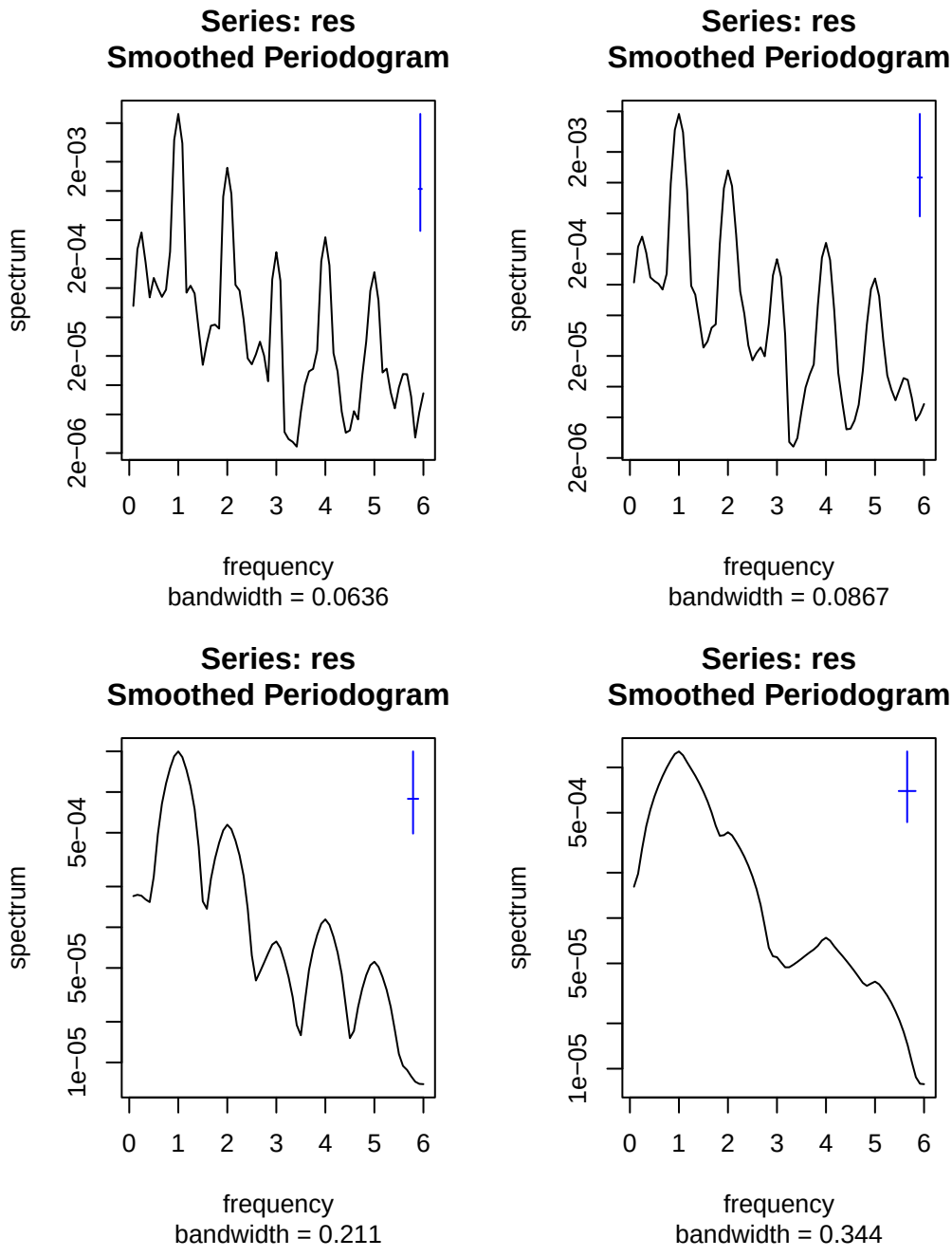


Figure 15.2.: Smoothed periodograms of monthly log-transformed and detrended `AirPassengers` time series.

Another method of testing the reality of a peak is to look at its harmonics. It is extremely unlikely that a true cycle will be shaped perfectly as a sine curve, hence at least a few of the first harmonics will show up as well. For example, in monthly data with annual seasonality (period of 12 months), we often observe peaks at 6, 4, and 3 months. This is exactly the pattern that we see in the figures above.
We can also approximate our data with an AR model and then plot the approximating periodogram of the AR model (Figure 19.3).

```
spectrum(res, method = "ar")
```

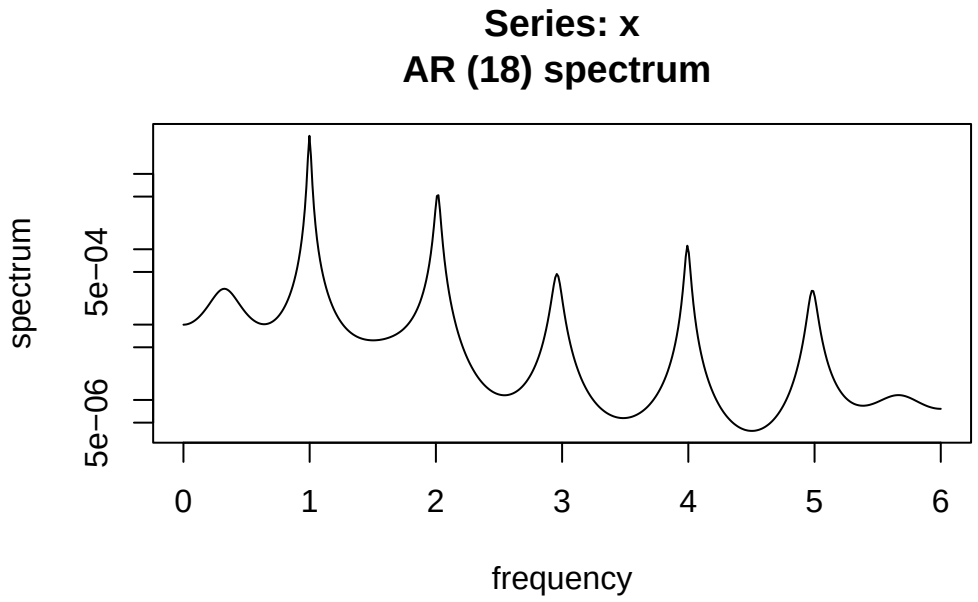


Figure 15.3.: Periodogram of AR process approximating the monthly log-transformed and detrended `AirPassengers` time series.

15.5. Periodogram for unequally-spaced time series

Unequally/unevenly-spaced time series or gaps in observations (missing data) can be handled with the Lomb–Scargle periodogram calculation. This method was originally developed for the analysis of unevenly-spaced astronomical signals (Lomb 1976; Scargle 1982) but found application in many other domains, including environmental science (e.g., Ruf 1999; Campos et al. 2008; Siskey et al. 2016).

Example: Ammonium concentration in wastewater system

For example, consider a time series `imputeTS::tsNH4` of ammonium (NH_4) concentration in a wastewater system (Figure 19.4). This time series contains observations from regular 10-minute

intervals, but some of the observations are missing.

```
forecast::autoplot(imputeTS::tsNH4) +
    xlab("Day") +
    ylab(bquote(NH[4]~concentration))
```

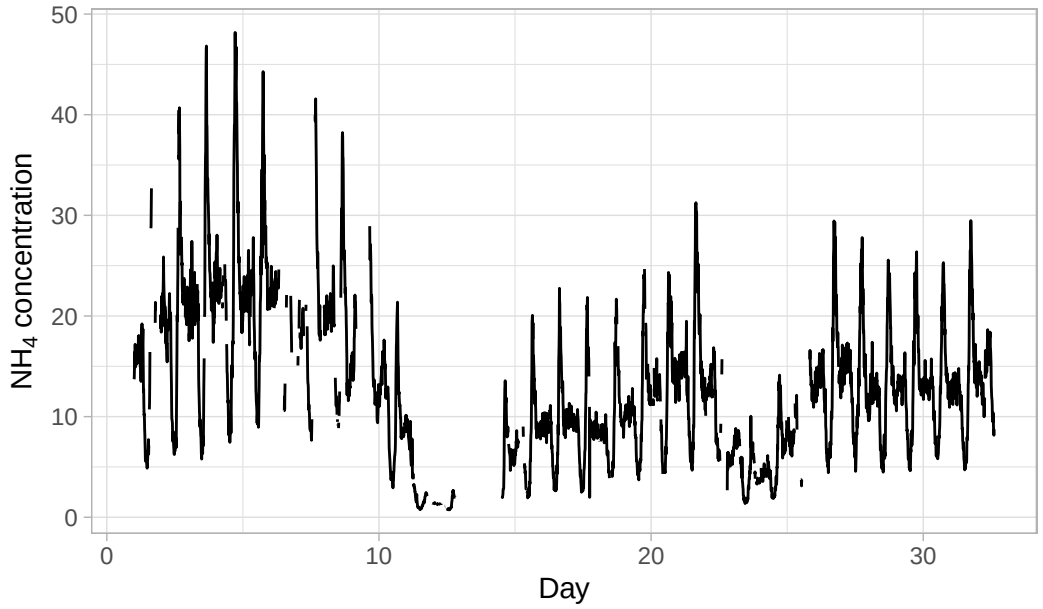


Figure 15.4.: Time series of ammonium concentration in a wastewater system.

Save the data in a format acceptable by the Lomb–Scargle function.

```
D <- data.frame(NH4 = as.vector(imputeTS::tsNH4),
                Time = as.vector(time(imputeTS::tsNH4)))
```

If needed, `na.omit(D)` can be used to remove the rows with missing values.
Figure 19.5 and Figure 19.6 show periodograms calculated based on these data. Note that the function `lomb::lsp()` has arguments adjusting the span of the frequencies and significance level.

```
par(mfrow = c(2, 2))
lomb::lsp(D$NH4, times = D$Time, type = "frequency", alpha = 0.05, main = "")
```

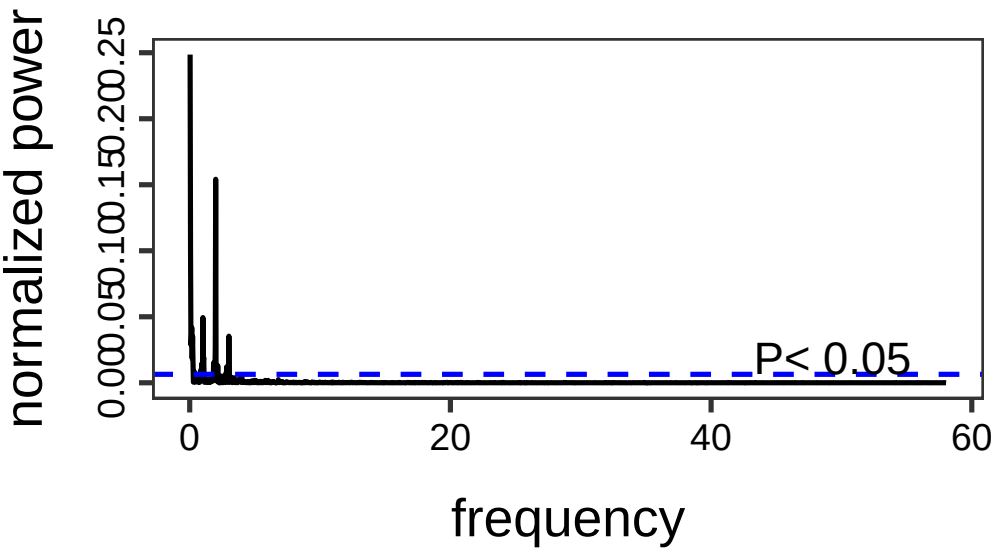


Figure 15.5.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

```
lomb::lsp(D$NH4, times = D$Time, type = "period", alpha = 0.05, main = "")
```

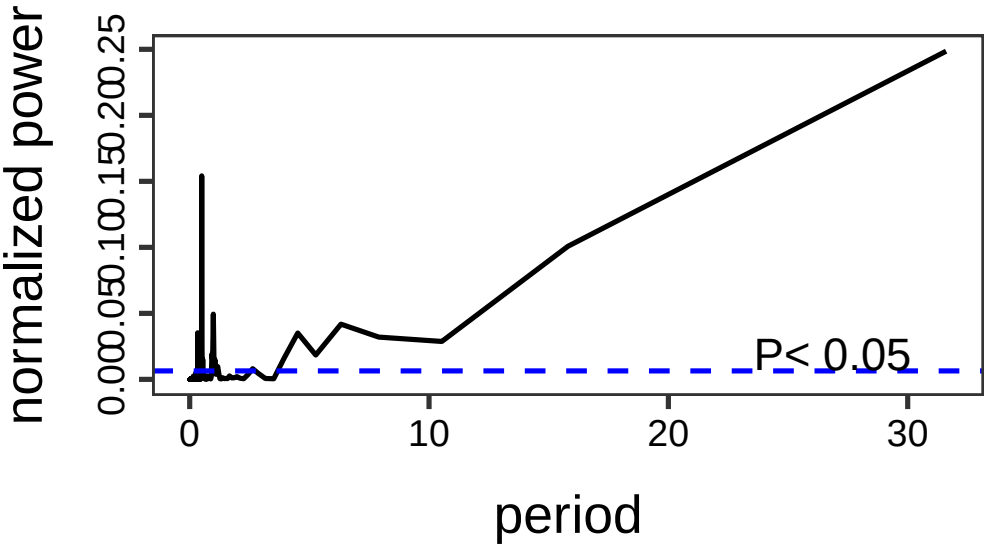


Figure 15.6.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

15.6. Frequency estimation in time

The assumption of a whole time series having a constant spectrum might be very limiting for the analysis of long time series. An estimation of frequencies locally in time allows us to study how the spectrum changes in time, and also detect smaller (short-lived) signals that would be averaged out otherwise. A straightforward solution is to separate the time series into windows (sub-periods) and estimate a spectrum in each such window.

The windowed analysis makes sense when in each window we have enough observations to estimate the frequencies. As a guide, we should have at least two observations per period to be able to represent the signal in the discrete Fourier transform. For example, we cannot estimate the seasonality if we sample once per year – this is our *sampling rate* or *sampling frequency* F_s. To start identifying seasonal periodicity, our sampling rate should be at least 2 samples per year. The highest frequency we can work with is limited by the *Nyquist frequency*

$$F_N = \frac{F_s}{2},$$

which is half of the sampling frequency.

After applying the FFT in each window, the results can be visualized in a *spectrogram*. The spectrogram combines information from multiple periodograms: it shows time on the x-axis, frequency on the y-axis, and the corresponding spectrum power as the color.

Example: Spectrograms for the NAO

Consider a long time series of the North Atlantic Oscillation (NAO) index obtained from an ensemble of 100 model-constrained NAO reconstructions (Figure 19.7).

```
NAOdf <- readr::read_csv("data/NAO.csv", skip = 12, show_col_types = FALSE) %>%
    rename(NAOproxy = nao_mc_mean) %>%
    select(Year, NAOproxy) %>%
    arrange(Year)
NAO <- ts(NAOdf$NAOproxy, start = NAOdf$Year[1])
spec_raw <- spec.pgram(NAO, detrend = FALSE, plot = FALSE, spans = 3)

# Find the frequency with the max power
fmax <- spec_raw$freq[which.max(spec_raw$spec)]
```

```
p1 <- forecast::autoplot(NAO) +
    xlab("Year") +
    ylab("NAO")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p3 <- data.frame(Spectrum = spec_raw$spec,
                 Period = 1/spec_raw$freq) %>%
    ggplot(aes(x = Period, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = 1/fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

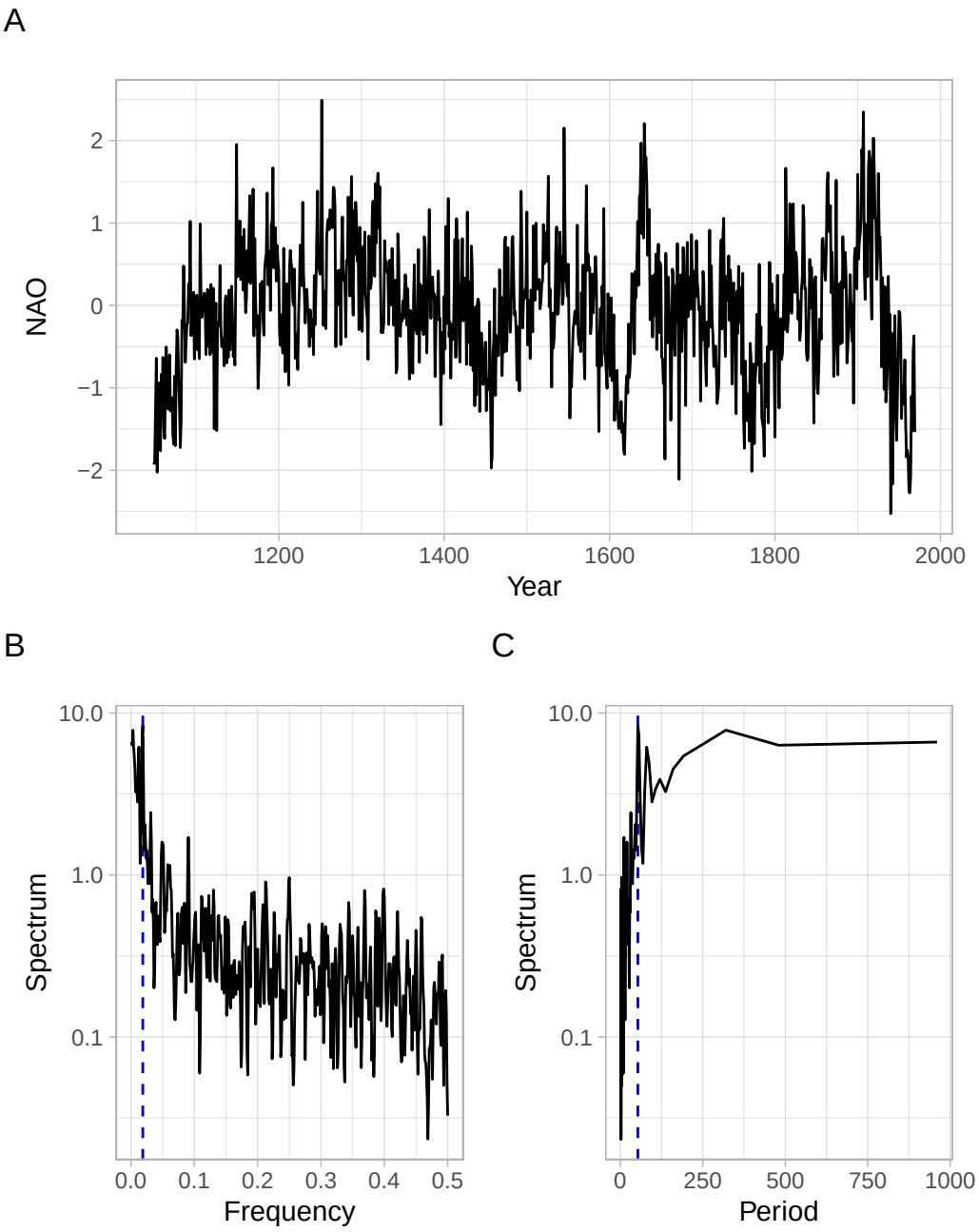


Figure 15.7.: North Atlantic Oscillation (NAO) winter index (December, January, and February) calculated as the mean of 100 model-constrained NAO reconstructions, along with its smoothed periodograms. The dashed lines denote the maximal magnitude of the spectrum.

This is an annual time series, hence the sampling frequency $F_s = 1$. From the global FFT results, the frequency with the maximal power is 0.019 year^{-1}, which is equivalent to the period of about 53.3 years. We will set the window for the FFT and calculate the spectrogram. Note

that we can use overlapping windows for a smoother picture.

```
# Set the window size (years), for applying the FFT in each window
window_size = 30

# Compute the spectrogram
SP <- signal::specgram(x = NAO,
                       n = window_size,
                       overlap = window_size/3,
                       Fs = 1)
```

See the spectrograms in Figure 19.8 and compare them with the time series plot in Figure 19.7 A.

```
par(mfrow = c(1, 2))

# Plot the spectrogram without decorations
print(SP, main = "A) Raw results")

# Discard phase information
P <- abs(SP$S)

# Normalize the spectrum to the range [0, 1]
P <- P - min(P)
P <- P/max(P)

# Extract time
Years <- time(NAO)[SP$t]

# Plot the spectrogram after processing
oce::imagep(x = Years,
            y = SP$f,
            z = t(P),
            col = oce::oce.colorsViridis,
            ylab = expression(Frequency~(y^-1 )),
            xlab = "Year",
            main = "B) After normalization and adding colors",
            drawPalette = TRUE,
            decimate = FALSE)
```

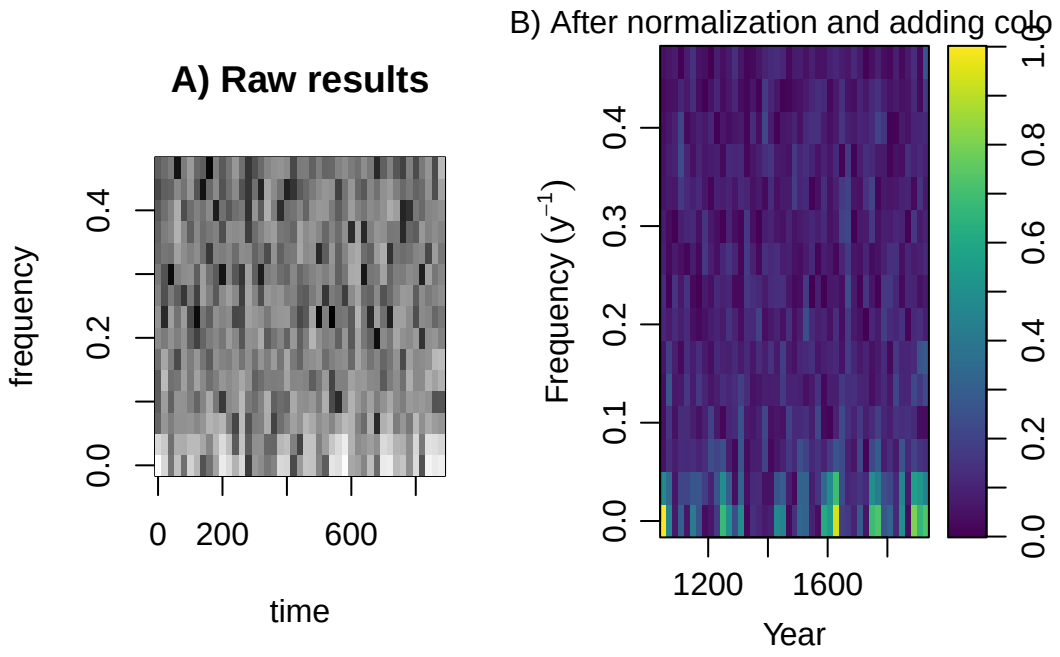


Figure 15.8.: Spectrograms for North Atlantic Oscillation (NAO) winter index.

15.7. Conclusion

For challenging problems, smoothing, multitapering, linear filtering, (repeated) pre-whitening, and Lomb–Scargle can be used together. Beware that aperiodic but autoregressive processes produce peaks in spectral densities. Harmonic analysis is a complicated art rather than a straightforward procedure.

It is extremely difficult to derive the significance of a weak periodicity from harmonic analysis. Do not believe analytical estimates (e.g., exponential probability), as they rarely apply to real data. It is essential to make simulations, typically permuting or bootstrapping the data keeping the observing times fixed. Simulations of the final model with the observation times are also advised.

15.8. Appendix

Wavelet analysis

An alternative to the windowed FFT is the wavelet analysis, which also estimates the strength of time series variations for different frequencies and times. *Wavelet* is a 'small wave' or function $\psi(t)$ approximating the variations. Popular wavelet functions are Meyer, Morlet, and Mexican hat.

Figure 19.9 shows the results of using the Morlet wavelet to process the NAO time series.

```
library(dplR)
wv <- morlet(y1 = NAOdf$NAOproxy, x1 = NAOdf$Year)
levs <- quantile(wv$Power, probs = c(0, 0.5, 0.75, 0.9, 0.95, 1))
wavelet.plot(wv, wavelet.levels = levs)
```

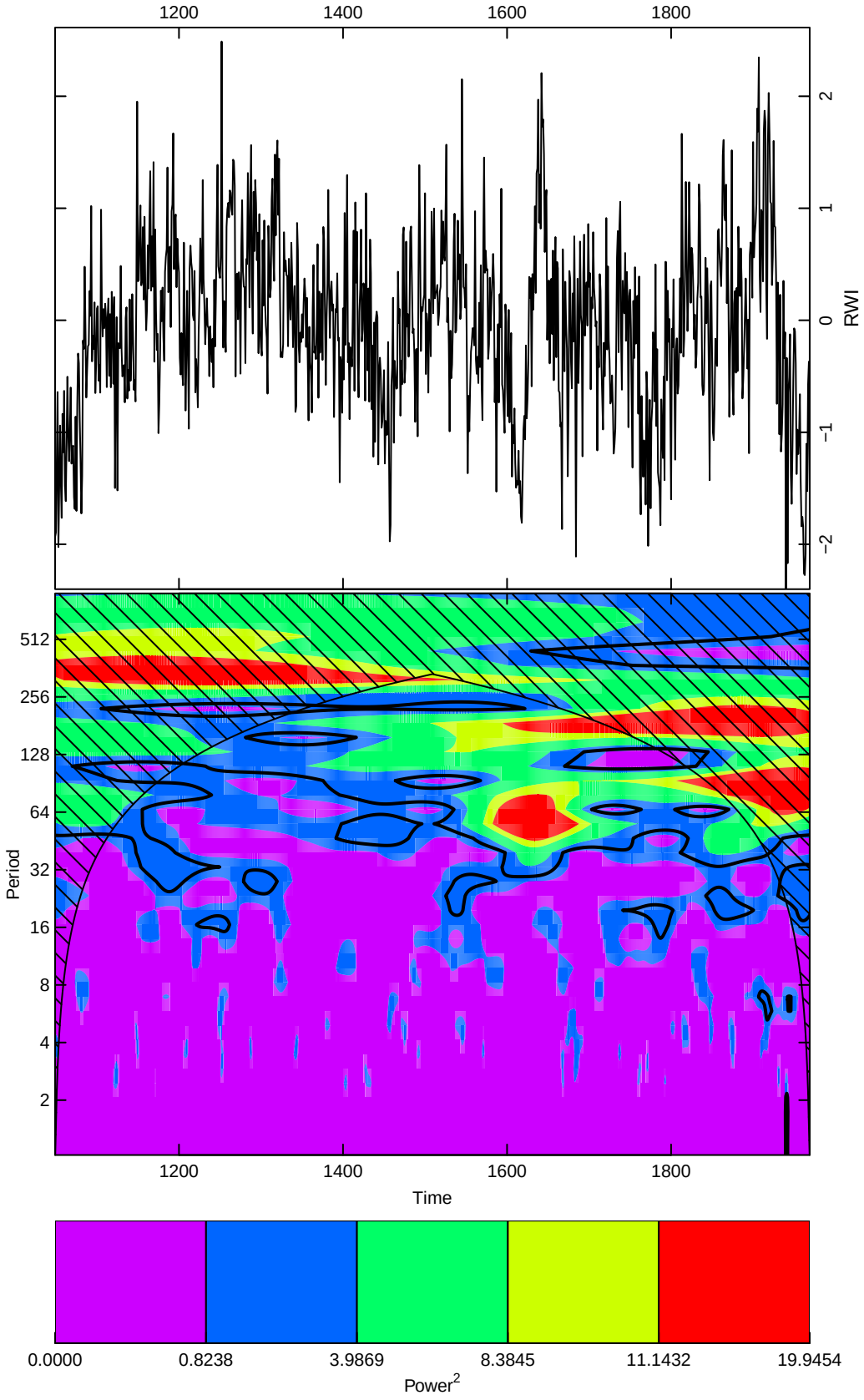


Figure 15.9.: Wavelet analysis of the North Atlantic Oscillation (NAO) winter index.

Part VIII.

Module 8 - Research Design & Evaluation

16. Model Evaluation and Analysis

This chapter provides a unified framework for evaluating and analyzing models across multiple AI paradigms: `Machine Learning (ML)`, `Deep Learning (DL)`, `Deep Reinforcement Learning (DRL)`, `Large Language Models (LLMs)`, and `Computer Vision (CV)`. The goal is to assess performance, robustness, efficiency, and interpretability, providing actionable insights for model improvement and real-world deployment.

16.1. ML Model Evaluation & Analysis

> Applies to classical models like Decision Trees, Random Forests, SVMs, and Gradient Boosting for structured data.

The Evaluation Metrics are :

16.1.1. Calculate Mean Absolute Error (MAE)

Analysis

- Use cross-validation for unbiased performance estimation.
- Visualize confusion matrix and feature importances.
- Evaluate bias–variance tradeoff with learning curves.
- Assess scalability for increasing dataset sizes.

Interpretation: - Consistent performance across folds with stable feature importance indicates generalizatio

Part IX.

Module 9 - Trustworthy & Robust AI

17. Trustworthy & Secure AI

The goal of this lecture is to learn how to view time series from the frequency perspective. You should be able to recognize features of time series from a periodogram similar to how we did it using plots of the autocorrelation function (ACF).

Objectives

1. Describe the mechanics of Fourier transform for time series.
2. Present results of spectrum calculations using a periodogram or spectrogram.
3. Give examples of smoothed periodograms.
4. Demonstrate the use of the Lomb–Scargle periodogram for analysis of unequally-spaced time series.

Reading materials

- Chapter 4 in Shumway and Stoffer (2017)
- Nason (2008) on wavelet analysis

17.1. Introduction

By now, we have been working in the *time domain*, such that our analysis could be seen as a regression of the present on the past (for example, ARIMA models). We have been using many time series plots with time on the x-axis.

An alternative approach is to analyze time series in a *spectral domain*, such that use a regression of present on a linear combination of sine and cosine functions. In this type of analysis, we often use periodogram plots, with frequency or period on the x-axis.

17.2. Regression on sinusoidal components

The simplest form of spectral analysis consists of regression on a periodic component:

$$Y_t = A\cos\omega t + B\sin\omega t + C + \epsilon_t, \tag{17.1}$$

where $t = 1, \dots, T$ and $\epsilon_t \sim \text{WN}(0, \sigma^2)$. Without loss of generality, we assume $0 \leqslant \omega \leqslant \pi$. In fact, for discrete data, frequencies outside this range are aliased into this range. For example, suppose that $-\pi < (\omega = \pi - \delta) < 0$, then

$$\begin{aligned}\cos(\omega t) &= \cos((\pi - \delta)t) \\ &= \cos(\pi t)\cos(\delta t) + \sin(\pi t)\sin(\delta t) \\ &= \cos(\delta t).\end{aligned}$$

Hence, a sampled sinusoid with a frequency smaller than 0 appears to coincide with a sinusoid with frequency in the interval $[0, \pi]$.

> **Note**
>
> The term $A\cos\omega t + B\sin\omega t$ is a periodic function with the period $2\pi/\omega$. The period $2\pi/\omega$ represents the number of time units that it takes for the function to take the same value again, i.e., to complete a cycle. The frequency, measured in cycles per time unit, is given by the inverse $\omega/(2\pi)$. The angular frequency, measured in radians per time unit, is given by ω. Because of its convenience, the angular frequency ω will be used to describe the periodicity of the function, and its name is shortened to frequency when there is no danger of confusion.

Consider monthly data that exhibit a 12-month seasonality. Hence, the period $2\pi/\omega = 12$, which implies the angular frequency $\omega = \pi/6$. The frequency, measured in cycles per time unit, is given by the inverse

$$\frac{\omega}{2\pi} = \frac{1}{12} \approx 0.08.$$

The formulas to estimate parameters of regression in Equation 19.1 take a much simpler form if ω is one of the Fourier frequencies, defined by

$$\omega_j = \frac{2\pi j}{T}, \quad j = 0, \ldots, \frac{T}{2},$$

then

$$\begin{aligned}
\hat{A} &= \frac{2}{T}\sum_t Y_t \cos\omega_j t, \\
\hat{B} &= \frac{2}{T}\sum_t Y_t \sin\omega_j t, \\
\hat{C} &= \overline{Y} = \frac{1}{T}\sum_t Y_t.
\end{aligned}$$

A suitable way of testing the significance of the sinusoidal component with frequency ω_j is using its contribution to the sum of squares

$$R_T(\omega_j) = \frac{T}{2}\left(\hat{A}^2 + \hat{B}^2\right).$$

If the $\epsilon_t \sim N(0, \sigma^2)$, then it follows that $\hat{A}$ and $\hat{B}$ are also independent normal, each with the variance $2\sigma^2/T$, so under the null hypothesis of $A = B = 0$ we find that

$$\frac{R_T(\omega_j)}{\sigma^2} \sim \chi^2_2$$

or equivalently that $R_T(\omega_j)/(2\sigma^2)$ has an exponential distribution with mean 1. The above theory can be extended to the simultaneous estimation of several periodic components.

17.3. Periodogram

The *Fourier transform* uses Fourier series, such as the pairs of sines and cosines with different periods, to describe the frequencies present in the original time series.

The Fourier transform applied to an equally-spaced time series Y_t (where $t = 1, \dots, T$) is also called the *discrete Fourier transform* (DFT) because the time is discrete (not the values Y_t).

To reduce the computational complexity of DFT and speed up the computations, one of the *fast Fourier transform* (FFT) algorithms is typically used.

The results of the Fourier transform are shown in a periodogram that describes the spectral properties of the signal.

The periodogram is defined as

$$I_T(\omega) = \frac{1}{2\pi T} \left| \sum_{t=1}^{T} Y_t e^{i\omega t} \right|^2 ,$$

which is an approximately unbiased estimator of the spectral density f.

Some undesirable features of the periodogram:

1. $I_T(\omega)$ for fixed ω is not a consistent estimate of $f(\omega)$, since

$$I_T(\omega_j) \sim \frac{f(\omega_j)}{2} \chi_2^2.$$

   Therefore, the variance of $f^2(\omega)$ does not tend to 0 as $T \to \infty$.
2. The independence of periodogram ordinates at different Fourier frequencies suggests that the sample periodogram plotted as a function of ω will be extremely irregular.

Suppose that $\gamma(h)$ is the autocovariance function of a stationary process and that $f(\omega)$ is the spectral density for the same process (h is the time lag and ω is the frequency). The autocovariance $\gamma(h)$ and the spectral density $f(\omega)$ are related:

$$\gamma(h) = \int_{-1/2}^{1/2} e^{2\pi i \omega h} f(\omega) d\omega,$$

and

$$f(\omega) = \sum_{h=-\infty}^{+\infty} \gamma(h) e^{-2\pi i \omega h}.$$

In the language of advanced calculus, the autocovariance and spectral density are Fourier transform pairs. These Fourier transform equations show that there is a direct link between the time domain representation and the frequency domain representation of a time series.

17.4. Smoothing

The idea behind smoothing is to take weighted averages over neighboring frequencies to reduce the variability associated with individual periodogram values. However, such an operation necessarily introduces some bias into the estimation procedure. Theoretical studies focus on the amount of smoothing that is required to obtain an optimum trade-off between bias and variance. In practice, this usually means that the choice of a kernel and amount of smoothing is somewhat subjective.

The main form of a smoothed estimator is given by

$$\hat{f}(\lambda) = \int_{-\pi}^{\pi} \frac{1}{h} K\left(\frac{\omega - \lambda}{h}\right) I_T(\omega) d\omega,$$

where $I_T(\cdot)$ is the periodogram based on T observations, $K(\cdot)$ is a kernel function, and h is the bandwidth. We usually take $K(\cdot)$ to be a nonnegative function, symmetric about 0 and integrating to 1. Thus, any symmetric density, such as the normal density, will work. In practice, however, it is more usual to take a kernel of finite range, such as the *Epanechnikov kernel*

$$K(x) = \frac{3}{4\sqrt{5}}\left(1 - \frac{x^2}{5}\right), \quad -\sqrt{5} \leqslant x \leqslant \sqrt{5},$$

which is 0 outside the interval $[-\sqrt{5}, \sqrt{5}]$. This choice of kernel function has some optimality properties. However, in practice, this optimality is less important than the choice of bandwidth h, which effectively controls the range over which the periodogram is smoothed.

There are some additional difficulties with the performance of the sample periodogram in the presence of a sinusoidal variation whose frequency is not one of the Fourier frequencies. This effect is known as *leakage*. The reason of leakage is that we always consider a truncated periodogram. Truncation implicitly assumes that the time series is periodic with a period T, which, of course, is not always true. So we artificially introduce non-existent periodicities into the estimated spectrum, i.e., cause 'leakage' of the spectrum. (If the time series is perfectly periodic over T then there is no leakage.) The leakage can be treated using an operation of tapering on the periodogram, i.e., by choosing appropriate periodogram windows.

> **Note**
>
> When we work with periodograms, we lose all phase (relative location/time origin) information: the periodogram will be the same if all the data were circularly rotated to a new time origin, i.e., the observed data are treated as perfectly periodic.

Another popular choice to smooth a periodogram is the *Daniell kernel* (with parameter m). For a time series, it is a centered moving average of values between the times $t - m$ and $t + m$ (inclusive). For example, the smoothing formula for a Daniell kernel with $m = 2$ is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2}}{5} \\ &= 0.2x_{t-2} + 0.2x_{t-1} + 0.2x_t + 0.2x_{t+1} + 0.2x_{t+2}.\end{aligned}$$

The weighting coefficients for a Daniell kernel with $m = 2$ can be checked using the function `kernel()`. In the output, the indices in `coef[ ]` refer to the time difference from the center of the average at the time t.

```
kernel("daniell", m = 2)
```

```
#> Daniell(2)
#> coef[-2] = 0.2
#> coef[-1] = 0.2
#> coef[ 0] = 0.2
#> coef[ 1] = 0.2
#> coef[ 2] = 0.2
```

The modified Daniell kernel is such that the two endpoints in the averaging receive half the weight that the interior points do. For a modified Daniell kernel with $m = 2$, the smoothing is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + 2x_{t-1} + 2x_t + 2x_{t+1} + x_{t+2}}{8} \\ &= 0.125x_{t-2} + 0.25x_{t-1} + 0.25x_t + 0.25x_{t+1} + 0.125x_{t+2}\end{aligned}$$

List the weighting coefficients:

```
kernel("modified.daniell", m = 2)
```

```
#> mDaniell(2)
#> coef[-2] = 0.125
#> coef[-1] = 0.250
#> coef[ 0] = 0.250
#> coef[ 1] = 0.250
#> coef[ 2] = 0.125
```

Either the Daniell kernel or the modified Daniell kernel can be convoluted (repeated) so that the smoothing is applied again to the smoothed values. This produces a more extensive smoothing by averaging over a wider time interval. For instance, to repeat a Daniell kernel with $m = 2$ on the smoothed values that resulted from a Daniell kernel with $m = 2$, the formula would be

$$\hat{\hat{x}}_t = \frac{\hat{x}_{t-2} + \hat{x}_{t-1} + \hat{x}_t + \hat{x}_{t+1} + \hat{x}_{t+2}}{5}.$$

The weights for averaging the original data for convoluted Daniell kernels with $m = 2$ in both smooths will be

```
kernel("daniell", m = c(2, 2))
```

```
#> Daniell(2,2)
#> coef[-4] = 0.04
#> coef[-3] = 0.08
#> coef[-2] = 0.12
#> coef[-1] = 0.16
#> coef[ 0] = 0.20
```

```
#> coef[ 1] = 0.16
#> coef[ 2] = 0.12
#> coef[ 3] = 0.08
#> coef[ 4] = 0.04
```

```
kernel("modified.daniell", m = c(2, 2))
```

```
#> mDaniell(2,2)
#> coef[-4] = 0.0156
#> coef[-3] = 0.0625
#> coef[-2] = 0.1250
#> coef[-1] = 0.1875
#> coef[ 0] = 0.2188
#> coef[ 1] = 0.1875
#> coef[ 2] = 0.1250
#> coef[ 3] = 0.0625
#> coef[ 4] = 0.0156
```

The center values are weighted slightly more heavily in the modified Daniell kernel than in the unmodified one.

When we smooth a periodogram, we are smoothing *across frequencies* rather than across times.

Example: Spectral analysis of the airline passenger data

Recall the airline passenger time series (**?@fig-airpassangers**) that has an increasing trend and strong multiplicative seasonality.
The series should be detrended before a spectral analysis. For demonstration purposes, compute periodogram for the raw data and log-transformed and detrended.

```
# Fit a quadratic trend to log-transformed data
t <- as.vector(time(AirPassengers))
mod <- lm(log10(AirPassengers) ~ poly(t, degree = 2))
res <- ts(mod$residuals)
attributes(res) <- attributes(AirPassengers)

# Compute spectra
spec_raw <- spec.pgram(AirPassengers, detrend = FALSE, plot = FALSE)
spec_log_detrend <- spec.pgram(res, detrend = FALSE, plot = FALSE)
```

The x-axes of the periodograms correspond to $\omega/(2\pi)$ or the number of cycles per the time series period specified in the `ts` object (12 months). If the frequency of 12 was not specified for this time series, the x-axes would be different, and the first spectrum peak would correspond to ≈ 0.08.
Figure 19.1 B shows that the spectrum of the raw data is dominated by low frequencies (trend). After detrending the data (Figure 19.1 C), the spectrum at frequency 1 (meaning one cycle per year) is the dominant one.

```
pAirPassengers <- forecast::autoplot(AirPassengers, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers (thousand)") +
    ggtitle("Raw series")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
p3 <- forecast::autoplot(res, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers residuals (lg[thousand])") +
    ggtitle("Detrended and log-transformed series")
p4 <- data.frame(Spectrum = spec_log_detrend$spec,
                 Frequency = spec_log_detrend$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
(pAirPassengers + p2) / (p3 + p4) +
    plot_annotation(tag_levels = 'A')
```

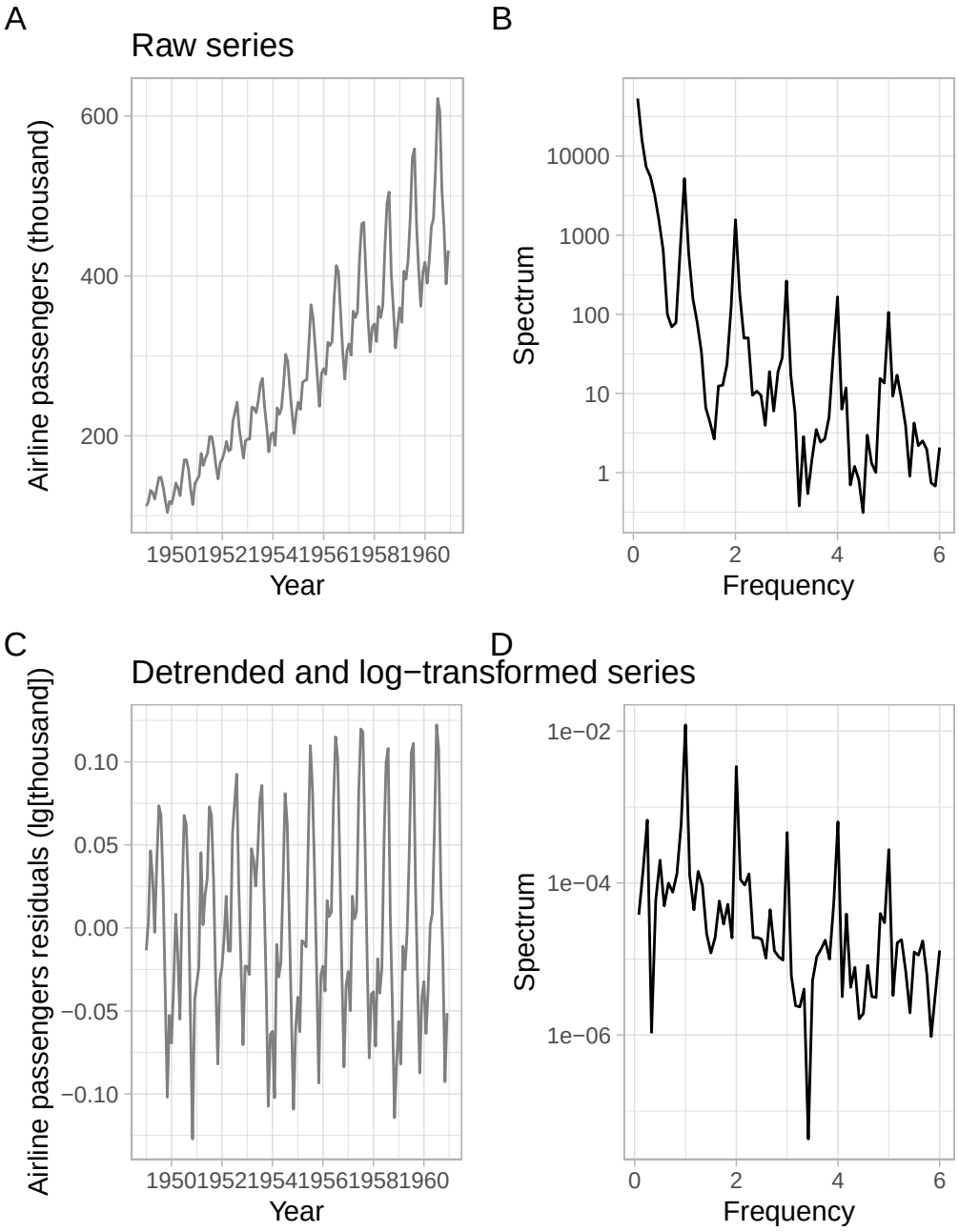


Figure 17.1.: Monthly `AirPassengers` time series, log-transformed and detrended, and the corresponding fast Fourier transforms.

Note that the function `spec.pgram()` has the option of removing a simpler (linear) trend and showing the results as a base-R plot. The confidence band in the upper right corner of this plot helps to identify the statistical significance of the peaks (Figure 19.2).

```
par(mfrow = c(2, 2))
spec.pgram(res, spans = c(3))
spec.pgram(res, spans = c(3, 3))
spec.pgram(res, spans = c(7, 7))
spec.pgram(res, spans = c(11, 11))
```

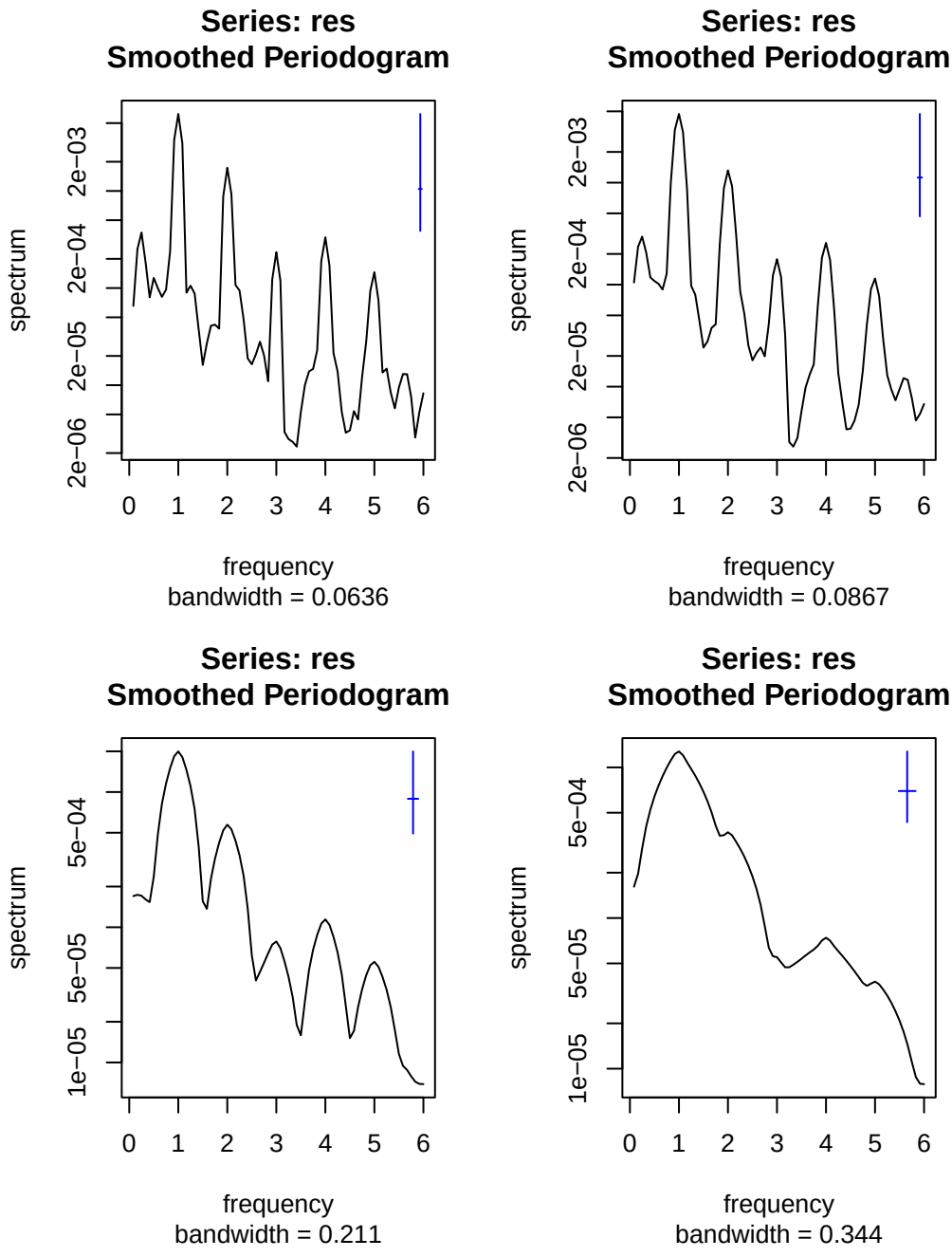


Figure 17.2.: Smoothed periodograms of monthly log-transformed and detrended `AirPassengers` time series.

Another method of testing the reality of a peak is to look at its harmonics. It is extremely unlikely that a true cycle will be shaped perfectly as a sine curve, hence at least a few of the first harmonics will show up as well. For example, in monthly data with annual seasonality (period of 12 months), we often observe peaks at 6, 4, and 3 months. This is exactly the pattern that we see in the figures above.
We can also approximate our data with an AR model and then plot the approximating periodogram of the AR model (Figure 19.3).

```
spectrum(res, method = "ar")
```

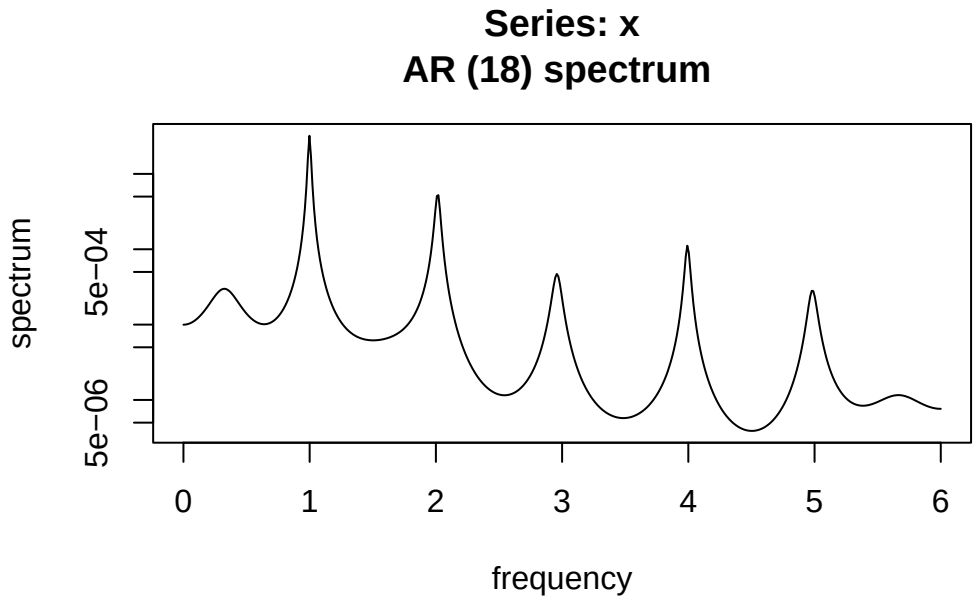


Figure 17.3.: Periodogram of AR process approximating the monthly log-transformed and detrended `AirPassengers` time series.

17.5. Periodogram for unequally-spaced time series

Unequally/unevenly-spaced time series or gaps in observations (missing data) can be handled with the Lomb–Scargle periodogram calculation. This method was originally developed for the analysis of unevenly-spaced astronomical signals (Lomb 1976; Scargle 1982) but found application in many other domains, including environmental science (e.g., Ruf 1999; Campos et al. 2008; Siskey et al. 2016).

Example: Ammonium concentration in wastewater system

For example, consider a time series `imputeTS::tsNH4` of ammonium (NH_4) concentration in a wastewater system (Figure 19.4). This time series contains observations from regular 10-minute

intervals, but some of the observations are missing.

```
forecast::autoplot(imputeTS::tsNH4) +
    xlab("Day") +
    ylab(bquote(NH[4]~concentration))
```

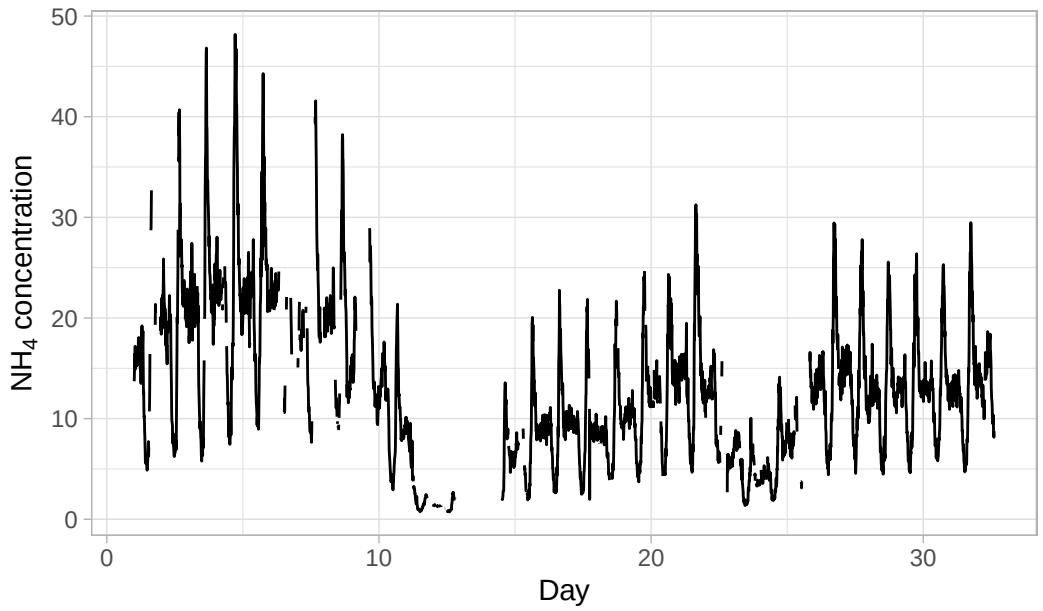


Figure 17.4.: Time series of ammonium concentration in a wastewater system.

Save the data in a format acceptable by the Lomb–Scargle function.

```
D <- data.frame(NH4 = as.vector(imputeTS::tsNH4),
                Time = as.vector(time(imputeTS::tsNH4)))
```

If needed, `na.omit(D)` can be used to remove the rows with missing values.
Figure 19.5 and Figure 19.6 show periodograms calculated based on these data. Note that the function `lomb::lsp()` has arguments adjusting the span of the frequencies and significance level.

```
par(mfrow = c(2, 2))
lomb::lsp(D$NH4, times = D$Time, type = "frequency", alpha = 0.05, main = "")
```

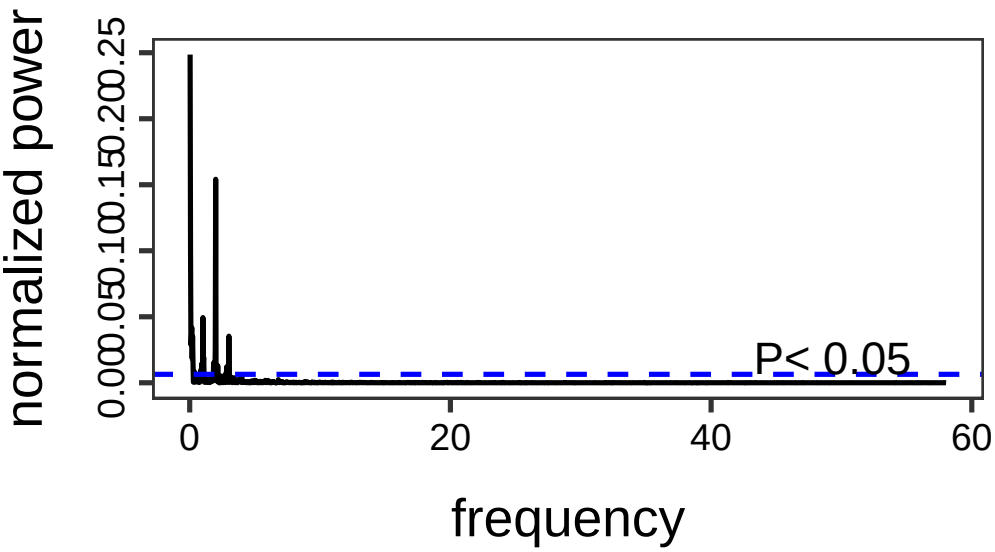


Figure 17.5.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

```
lomb::lsp(D$NH4, times = D$Time, type = "period", alpha = 0.05, main = "")
```

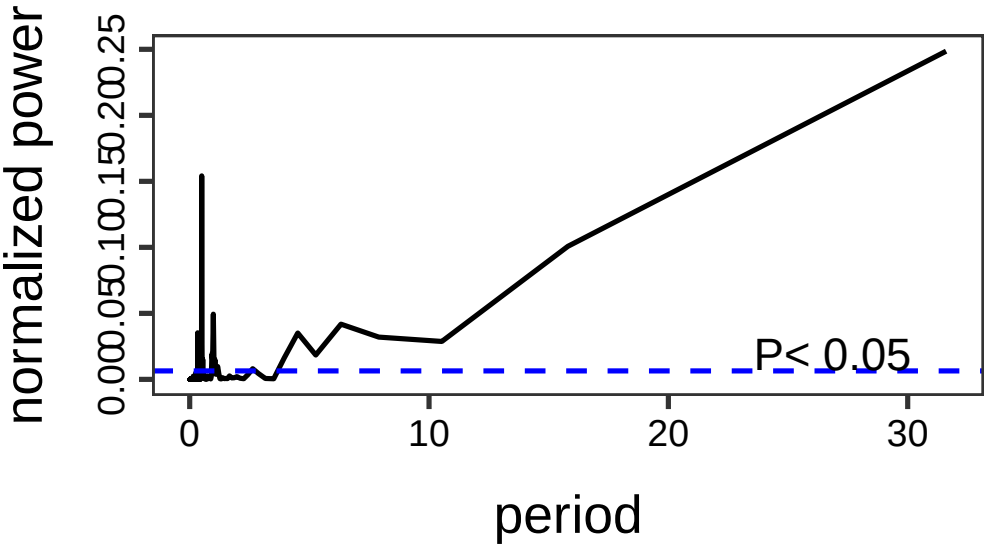


Figure 17.6.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

17.6. Frequency estimation in time

The assumption of a whole time series having a constant spectrum might be very limiting for the analysis of long time series. An estimation of frequencies locally in time allows us to study how the spectrum changes in time, and also detect smaller (short-lived) signals that would be averaged out otherwise. A straightforward solution is to separate the time series into windows (sub-periods) and estimate a spectrum in each such window.

The windowed analysis makes sense when in each window we have enough observations to estimate the frequencies. As a guide, we should have at least two observations per period to be able to represent the signal in the discrete Fourier transform. For example, we cannot estimate the seasonality if we sample once per year – this is our *sampling rate* or *sampling frequency* F_s. To start identifying seasonal periodicity, our sampling rate should be at least 2 samples per year. The highest frequency we can work with is limited by the *Nyquist frequency*

$$F_N = \frac{F_s}{2},$$

which is half of the sampling frequency.

After applying the FFT in each window, the results can be visualized in a *spectrogram*. The spectrogram combines information from multiple periodograms: it shows time on the x-axis, frequency on the y-axis, and the corresponding spectrum power as the color.

Example: Spectrograms for the NAO

Consider a long time series of the North Atlantic Oscillation (NAO) index obtained from an ensemble of 100 model-constrained NAO reconstructions (Figure 19.7).

```
NAOdf <- readr::read_csv("data/NAO.csv", skip = 12, show_col_types = FALSE) %>%
    rename(NAOproxy = nao_mc_mean) %>%
    select(Year, NAOproxy) %>%
    arrange(Year)
NAO <- ts(NAOdf$NAOproxy, start = NAOdf$Year[1])
spec_raw <- spec.pgram(NAO, detrend = FALSE, plot = FALSE, spans = 3)

# Find the frequency with the max power
fmax <- spec_raw$freq[which.max(spec_raw$spec)]
```

```
p1 <- forecast::autoplot(NAO) +
    xlab("Year") +
    ylab("NAO")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p3 <- data.frame(Spectrum = spec_raw$spec,
                 Period = 1/spec_raw$freq) %>%
    ggplot(aes(x = Period, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = 1/fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

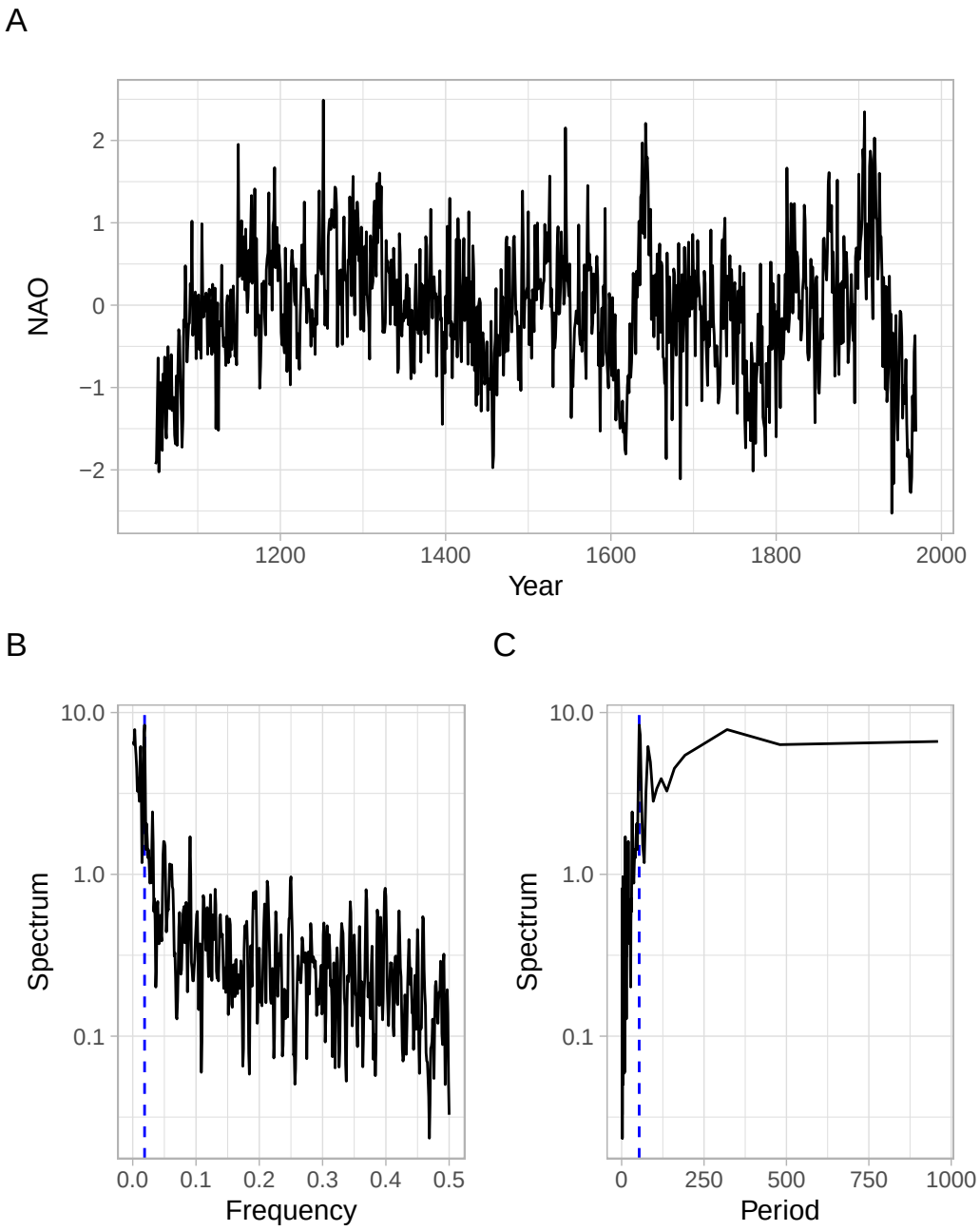


Figure 17.7.: North Atlantic Oscillation (NAO) winter index (December, January, and February) calculated as the mean of 100 model-constrained NAO reconstructions, along with its smoothed periodograms. The dashed lines denote the maximal magnitude of the spectrum.

This is an annual time series, hence the sampling frequency $F_s = 1$. From the global FFT results, the frequency with the maximal power is 0.019 year^{-1}, which is equivalent to the period of about 53.3 years. We will set the window for the FFT and calculate the spectrogram. Note

that we can use overlapping windows for a smoother picture.

```
# Set the window size (years), for applying the FFT in each window
window_size = 30

# Compute the spectrogram
SP <- signal::specgram(x = NAO,
                       n = window_size,
                       overlap = window_size/3,
                       Fs = 1)
```

See the spectrograms in Figure 19.8 and compare them with the time series plot in Figure 19.7 A.

```
par(mfrow = c(1, 2))

# Plot the spectrogram without decorations
print(SP, main = "A) Raw results")

# Discard phase information
P <- abs(SP$S)

# Normalize the spectrum to the range [0, 1]
P <- P - min(P)
P <- P/max(P)

# Extract time
Years <- time(NAO)[SP$t]

# Plot the spectrogram after processing
oce::imagep(x = Years,
            y = SP$f,
            z = t(P),
            col = oce::oce.colorsViridis,
            ylab = expression(Frequency~(y^-1 )),
            xlab = "Year",
            main = "B) After normalization and adding colors",
            drawPalette = TRUE,
            decimate = FALSE)
```

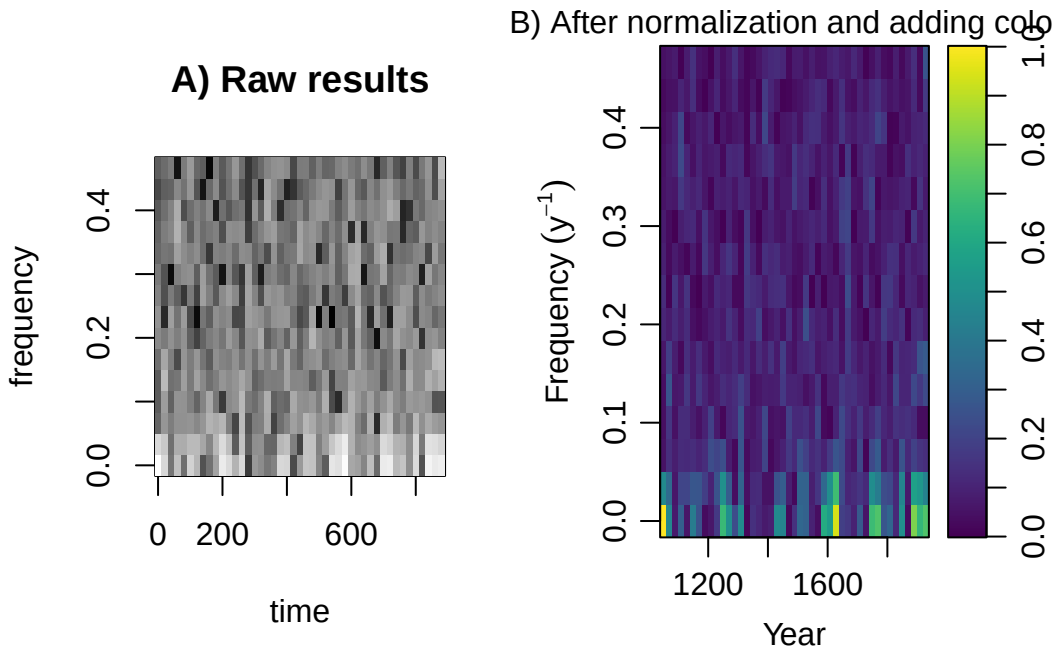


Figure 17.8.: Spectrograms for North Atlantic Oscillation (NAO) winter index.

17.7. Conclusion

For challenging problems, smoothing, multitapering, linear filtering, (repeated) pre-whitening, and Lomb–Scargle can be used together. Beware that aperiodic but autoregressive processes produce peaks in spectral densities. Harmonic analysis is a complicated art rather than a straightforward procedure.

It is extremely difficult to derive the significance of a weak periodicity from harmonic analysis. Do not believe analytical estimates (e.g., exponential probability), as they rarely apply to real data. It is essential to make simulations, typically permuting or bootstrapping the data keeping the observing times fixed. Simulations of the final model with the observation times are also advised.

17.8. Appendix

Wavelet analysis

An alternative to the windowed FFT is the wavelet analysis, which also estimates the strength of time series variations for different frequencies and times. *Wavelet* is a 'small wave' or function $\psi(t)$ approximating the variations. Popular wavelet functions are Meyer, Morlet, and Mexican hat.

Figure 19.9 shows the results of using the Morlet wavelet to process the NAO time series.

```
library(dplR)
wv <- morlet(y1 = NAOdf$NAOproxy, x1 = NAOdf$Year)
levs <- quantile(wv$Power, probs = c(0, 0.5, 0.75, 0.9, 0.95, 1))
wavelet.plot(wv, wavelet.levels = levs)
```

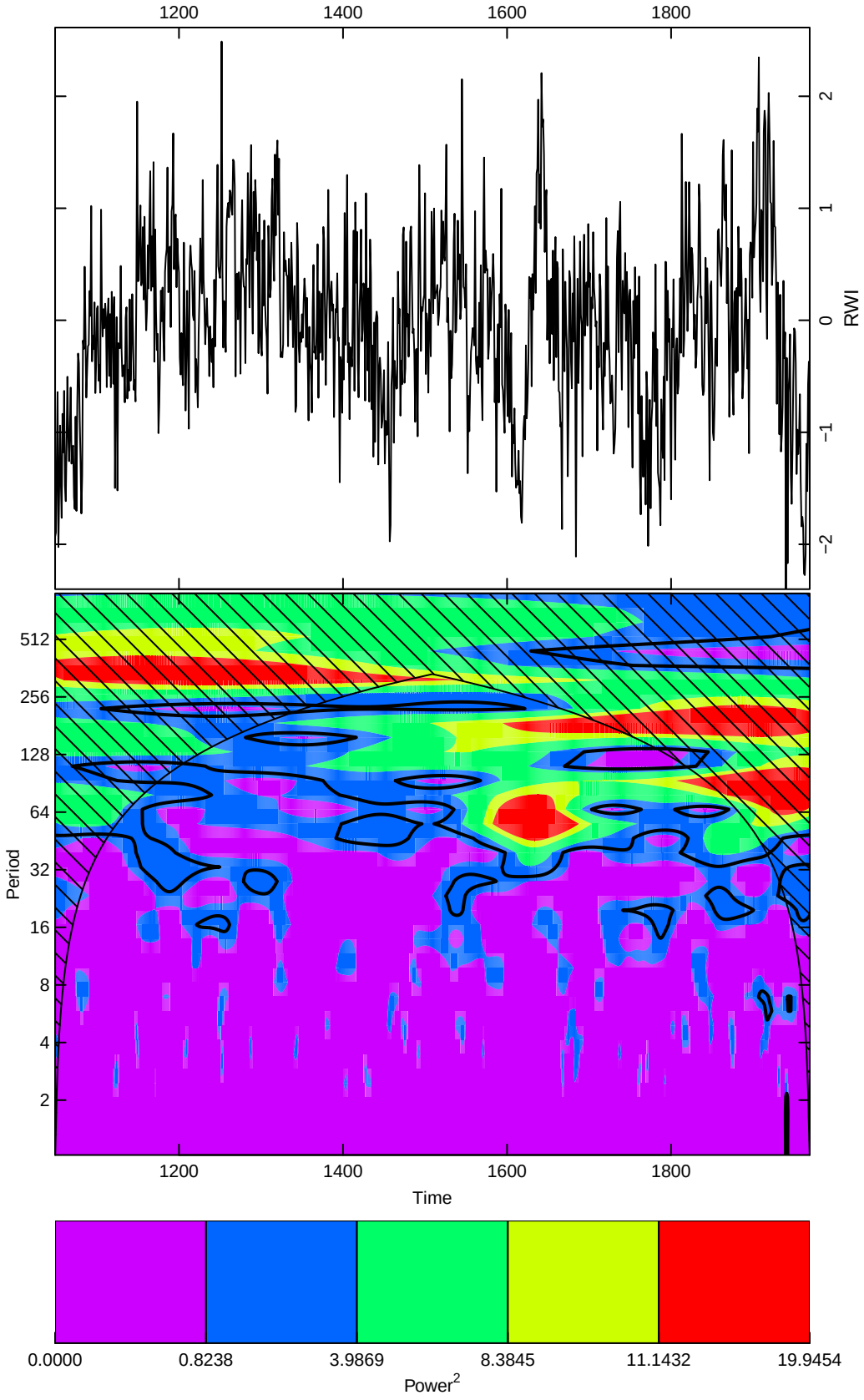


Figure 17.9.: Wavelet analysis of the North Atlantic Oscillation (NAO) winter index.

Part X.

Module 10 - Emerging Topics

18. Multimodal

The goal of this lecture is to learn how to view time series from the frequency perspective. You should be able to recognize features of time series from a periodogram similar to how we did it using plots of the autocorrelation function (ACF).

Objectives

1. Describe the mechanics of Fourier transform for time series.
2. Present results of spectrum calculations using a periodogram or spectrogram.
3. Give examples of smoothed periodograms.
4. Demonstrate the use of the Lomb–Scargle periodogram for analysis of unequally-spaced time series.

Reading materials

- Chapter 4 in Shumway and Stoffer (2017)
- Nason (2008) on wavelet analysis

18.1. Introduction

By now, we have been working in the *time domain*, such that our analysis could be seen as a regression of the present on the past (for example, ARIMA models). We have been using many time series plots with time on the x-axis.

An alternative approach is to analyze time series in a *spectral domain*, such that use a regression of present on a linear combination of sine and cosine functions. In this type of analysis, we often use periodogram plots, with frequency or period on the x-axis.

18.2. Regression on sinusoidal components

The simplest form of spectral analysis consists of regression on a periodic component:

$$Y_t = A\cos\omega t + B\sin\omega t + C + \epsilon_t, \tag{18.1}$$

where $t = 1, \dots, T$ and $\epsilon_t \sim \text{WN}(0, \sigma^2)$. Without loss of generality, we assume $0 \leqslant \omega \leqslant \pi$. In fact, for discrete data, frequencies outside this range are aliased into this range. For example, suppose that $-\pi < (\omega = \pi - \delta) < 0$, then

$$\begin{aligned}\cos(\omega t) &= \cos((\pi - \delta)t) \\ &= \cos(\pi t)\cos(\delta t) + \sin(\pi t)\sin(\delta t) \\ &= \cos(\delta t).\end{aligned}$$

Hence, a sampled sinusoid with a frequency smaller than 0 appears to coincide with a sinusoid with frequency in the interval $[0, \pi]$.

> **Note**
>
> The term $A\cos\omega t + B\sin\omega t$ is a periodic function with the period $2\pi/\omega$. The period $2\pi/\omega$ represents the number of time units that it takes for the function to take the same value again, i.e., to complete a cycle. The frequency, measured in cycles per time unit, is given by the inverse $\omega/(2\pi)$. The angular frequency, measured in radians per time unit, is given by ω. Because of its convenience, the angular frequency ω will be used to describe the periodicity of the function, and its name is shortened to frequency when there is no danger of confusion.

Consider monthly data that exhibit a 12-month seasonality. Hence, the period $2\pi/\omega = 12$, which implies the angular frequency $\omega = \pi/6$. The frequency, measured in cycles per time unit, is given by the inverse

$$\frac{\omega}{2\pi} = \frac{1}{12} \approx 0.08.$$

The formulas to estimate parameters of regression in Equation 19.1 take a much simpler form if ω is one of the Fourier frequencies, defined by

$$\omega_j = \frac{2\pi j}{T}, \quad j = 0, \dots, \frac{T}{2},$$

then

$$\begin{aligned}
\hat{A} &= \frac{2}{T}\sum_t Y_t \cos\omega_j t, \\
\hat{B} &= \frac{2}{T}\sum_t Y_t \sin\omega_j t, \\
\hat{C} &= \overline{Y} = \frac{1}{T}\sum_t Y_t.
\end{aligned}$$

A suitable way of testing the significance of the sinusoidal component with frequency ω_j is using its contribution to the sum of squares

$$R_T(\omega_j) = \frac{T}{2}\left(\hat{A}^2 + \hat{B}^2\right).$$

If the $\epsilon_t \sim N(0, \sigma^2)$, then it follows that $\hat{A}$ and $\hat{B}$ are also independent normal, each with the variance $2\sigma^2/T$, so under the null hypothesis of $A = B = 0$ we find that

$$\frac{R_T(\omega_j)}{\sigma^2} \sim \chi^2_2$$

or equivalently that $R_T(\omega_j)/(2\sigma^2)$ has an exponential distribution with mean 1. The above theory can be extended to the simultaneous estimation of several periodic components.

18.3. Periodogram

The *Fourier transform* uses Fourier series, such as the pairs of sines and cosines with different periods, to describe the frequencies present in the original time series.

The Fourier transform applied to an equally-spaced time series Y_t (where $t = 1, \dots, T$) is also called the *discrete Fourier transform* (DFT) because the time is discrete (not the values Y_t).

To reduce the computational complexity of DFT and speed up the computations, one of the *fast Fourier transform* (FFT) algorithms is typically used.

The results of the Fourier transform are shown in a periodogram that describes the spectral properties of the signal.

The periodogram is defined as

$$I_T(\omega) = \frac{1}{2\pi T}\left|\sum_{t=1}^{T} Y_t e^{i\omega t}\right|^2,$$

which is an approximately unbiased estimator of the spectral density f.

Some undesirable features of the periodogram:

1. $I_T(\omega)$ for fixed ω is not a consistent estimate of $f(\omega)$, since

$$I_T(\omega_j) \sim \frac{f(\omega_j)}{2}\chi_2^2.$$

   Therefore, the variance of $f^2(\omega)$ does not tend to 0 as $T \to \infty$.
2. The independence of periodogram ordinates at different Fourier frequencies suggests that the sample periodogram plotted as a function of ω will be extremely irregular.

Suppose that $\gamma(h)$ is the autocovariance function of a stationary process and that $f(\omega)$ is the spectral density for the same process (h is the time lag and ω is the frequency). The autocovariance $\gamma(h)$ and the spectral density $f(\omega)$ are related:

$$\gamma(h) = \int_{-1/2}^{1/2} e^{2\pi i\omega h} f(\omega) d\omega,$$

and

$$f(\omega) = \sum_{h=-\infty}^{+\infty} \gamma(h) e^{-2\pi i\omega h}.$$

In the language of advanced calculus, the autocovariance and spectral density are Fourier transform pairs. These Fourier transform equations show that there is a direct link between the time domain representation and the frequency domain representation of a time series.

18.4. Smoothing

The idea behind smoothing is to take weighted averages over neighboring frequencies to reduce the variability associated with individual periodogram values. However, such an operation necessarily introduces some bias into the estimation procedure. Theoretical studies focus on the amount of smoothing that is required to obtain an optimum trade-off between bias and variance. In practice, this usually means that the choice of a kernel and amount of smoothing is somewhat subjective.

The main form of a smoothed estimator is given by

$$\hat{f}(\lambda) = \int_{-\pi}^{\pi} \frac{1}{h} K\left(\frac{\omega - \lambda}{h}\right) I_T(\omega) d\omega,$$

where $I_T(\cdot)$ is the periodogram based on T observations, $K(\cdot)$ is a kernel function, and h is the bandwidth. We usually take $K(\cdot)$ to be a nonnegative function, symmetric about 0 and integrating to 1. Thus, any symmetric density, such as the normal density, will work. In practice, however, it is more usual to take a kernel of finite range, such as the *Epanechnikov kernel*

$$K(x) = \frac{3}{4\sqrt{5}}\left(1 - \frac{x^2}{5}\right), \quad -\sqrt{5} \leqslant x \leqslant \sqrt{5},$$

which is 0 outside the interval $[-\sqrt{5}, \sqrt{5}]$. This choice of kernel function has some optimality properties. However, in practice, this optimality is less important than the choice of bandwidth h, which effectively controls the range over which the periodogram is smoothed.

There are some additional difficulties with the performance of the sample periodogram in the presence of a sinusoidal variation whose frequency is not one of the Fourier frequencies. This effect is known as *leakage*. The reason of leakage is that we always consider a truncated periodogram. Truncation implicitly assumes that the time series is periodic with a period T, which, of course, is not always true. So we artificially introduce non-existent periodicities into the estimated spectrum, i.e., cause 'leakage' of the spectrum. (If the time series is perfectly periodic over T then there is no leakage.) The leakage can be treated using an operation of tapering on the periodogram, i.e., by choosing appropriate periodogram windows.

> **Note**
>
> When we work with periodograms, we lose all phase (relative location/time origin) information: the periodogram will be the same if all the data were circularly rotated to a new time origin, i.e., the observed data are treated as perfectly periodic.

Another popular choice to smooth a periodogram is the *Daniell kernel* (with parameter m). For a time series, it is a centered moving average of values between the times $t - m$ and $t + m$ (inclusive). For example, the smoothing formula for a Daniell kernel with $m = 2$ is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2}}{5} \\ &= 0.2x_{t-2} + 0.2x_{t-1} + 0.2x_t + 0.2x_{t+1} + 0.2x_{t+2}.\end{aligned}$$

The weighting coefficients for a Daniell kernel with $m = 2$ can be checked using the function `kernel()`. In the output, the indices in `coef[ ]` refer to the time difference from the center of the average at the time t.

```
kernel("daniell", m = 2)
```

```
#> Daniell(2)
#> coef[-2] = 0.2
#> coef[-1] = 0.2
#> coef[ 0] = 0.2
#> coef[ 1] = 0.2
#> coef[ 2] = 0.2
```

The modified Daniell kernel is such that the two endpoints in the averaging receive half the weight that the interior points do. For a modified Daniell kernel with $m = 2$, the smoothing is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + 2x_{t-1} + 2x_t + 2x_{t+1} + x_{t+2}}{8} \\ &= 0.125x_{t-2} + 0.25x_{t-1} + 0.25x_t + 0.25x_{t+1} + 0.125x_{t+2}\end{aligned}$$

List the weighting coefficients:

```
kernel("modified.daniell", m = 2)
```

```
#> mDaniell(2)
#> coef[-2] = 0.125
#> coef[-1] = 0.250
#> coef[ 0] = 0.250
#> coef[ 1] = 0.250
#> coef[ 2] = 0.125
```

Either the Daniell kernel or the modified Daniell kernel can be convoluted (repeated) so that the smoothing is applied again to the smoothed values. This produces a more extensive smoothing by averaging over a wider time interval. For instance, to repeat a Daniell kernel with $m = 2$ on the smoothed values that resulted from a Daniell kernel with $m = 2$, the formula would be

$$\hat{\hat{x}}_t = \frac{\hat{x}_{t-2} + \hat{x}_{t-1} + \hat{x}_t + \hat{x}_{t+1} + \hat{x}_{t+2}}{5}.$$

The weights for averaging the original data for convoluted Daniell kernels with $m = 2$ in both smooths will be

```
kernel("daniell", m = c(2, 2))
```

```
#> Daniell(2,2)
#> coef[-4] = 0.04
#> coef[-3] = 0.08
#> coef[-2] = 0.12
#> coef[-1] = 0.16
#> coef[ 0] = 0.20
```

```
#> coef[ 1] = 0.16
#> coef[ 2] = 0.12
#> coef[ 3] = 0.08
#> coef[ 4] = 0.04
```

```
kernel("modified.daniell", m = c(2, 2))
```

```
#> mDaniell(2,2)
#> coef[-4] = 0.0156
#> coef[-3] = 0.0625
#> coef[-2] = 0.1250
#> coef[-1] = 0.1875
#> coef[ 0] = 0.2188
#> coef[ 1] = 0.1875
#> coef[ 2] = 0.1250
#> coef[ 3] = 0.0625
#> coef[ 4] = 0.0156
```

The center values are weighted slightly more heavily in the modified Daniell kernel than in the unmodified one.

When we smooth a periodogram, we are smoothing *across frequencies* rather than across times.

Example: Spectral analysis of the airline passenger data

Recall the airline passenger time series (**?@fig-airpassangers**) that has an increasing trend and strong multiplicative seasonality.
The series should be detrended before a spectral analysis. For demonstration purposes, compute periodogram for the raw data and log-transformed and detrended.

```
# Fit a quadratic trend to log-transformed data
t <- as.vector(time(AirPassengers))
mod <- lm(log10(AirPassengers) ~ poly(t, degree = 2))
res <- ts(mod$residuals)
attributes(res) <- attributes(AirPassengers)

# Compute spectra
spec_raw <- spec.pgram(AirPassengers, detrend = FALSE, plot = FALSE)
spec_log_detrend <- spec.pgram(res, detrend = FALSE, plot = FALSE)
```

The x-axes of the periodograms correspond to $\omega/(2\pi)$ or the number of cycles per the time series period specified in the `ts` object (12 months). If the frequency of 12 was not specified for this time series, the x-axes would be different, and the first spectrum peak would correspond to ≈ 0.08.
Figure 19.1 B shows that the spectrum of the raw data is dominated by low frequencies (trend). After detrending the data (Figure 19.1 C), the spectrum at frequency 1 (meaning one cycle per year) is the dominant one.

```
pAirPassengers <- forecast::autoplot(AirPassengers, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers (thousand)") +
    ggtitle("Raw series")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
p3 <- forecast::autoplot(res, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers residuals (lg[thousand])") +
    ggtitle("Detrended and log-transformed series")
p4 <- data.frame(Spectrum = spec_log_detrend$spec,
                 Frequency = spec_log_detrend$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
(pAirPassengers + p2) / (p3 + p4) +
    plot_annotation(tag_levels = 'A')
```

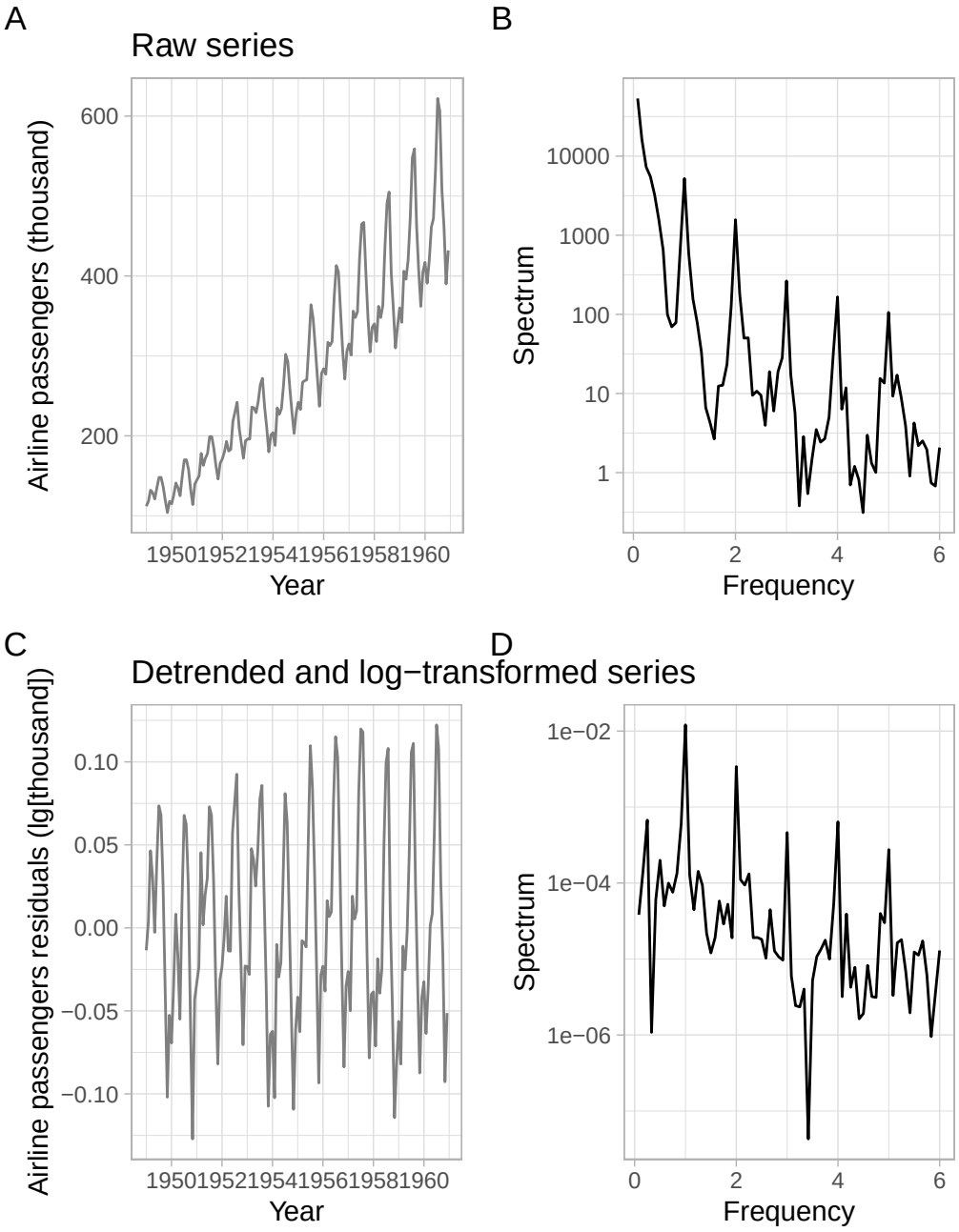


Figure 18.1.: Monthly `AirPassengers` time series, log-transformed and detrended, and the corresponding fast Fourier transforms.

Note that the function `spec.pgram()` has the option of removing a simpler (linear) trend and showing the results as a base-R plot. The confidence band in the upper right corner of this plot helps to identify the statistical significance of the peaks (Figure 19.2).

```r
par(mfrow = c(2, 2))
spec.pgram(res, spans = c(3))
spec.pgram(res, spans = c(3, 3))
spec.pgram(res, spans = c(7, 7))
spec.pgram(res, spans = c(11, 11))
```

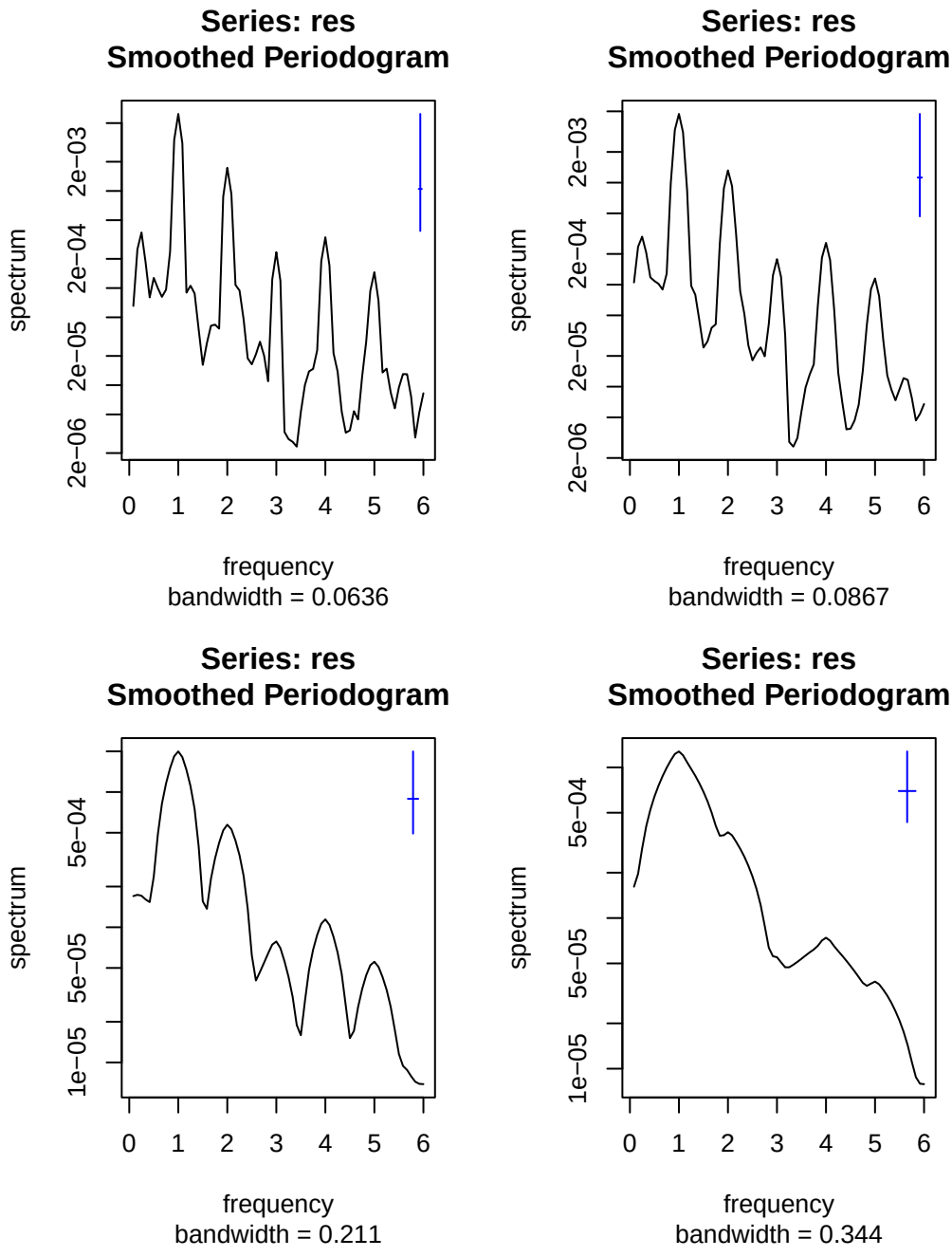


Figure 18.2.: Smoothed periodograms of monthly log-transformed and detrended `AirPassengers` time series.

Another method of testing the reality of a peak is to look at its harmonics. It is extremely unlikely that a true cycle will be shaped perfectly as a sine curve, hence at least a few of the first harmonics will show up as well. For example, in monthly data with annual seasonality (period of 12 months), we often observe peaks at 6, 4, and 3 months. This is exactly the pattern that we see in the figures above.
We can also approximate our data with an AR model and then plot the approximating periodogram of the AR model (Figure 19.3).

```
spectrum(res, method = "ar")
```

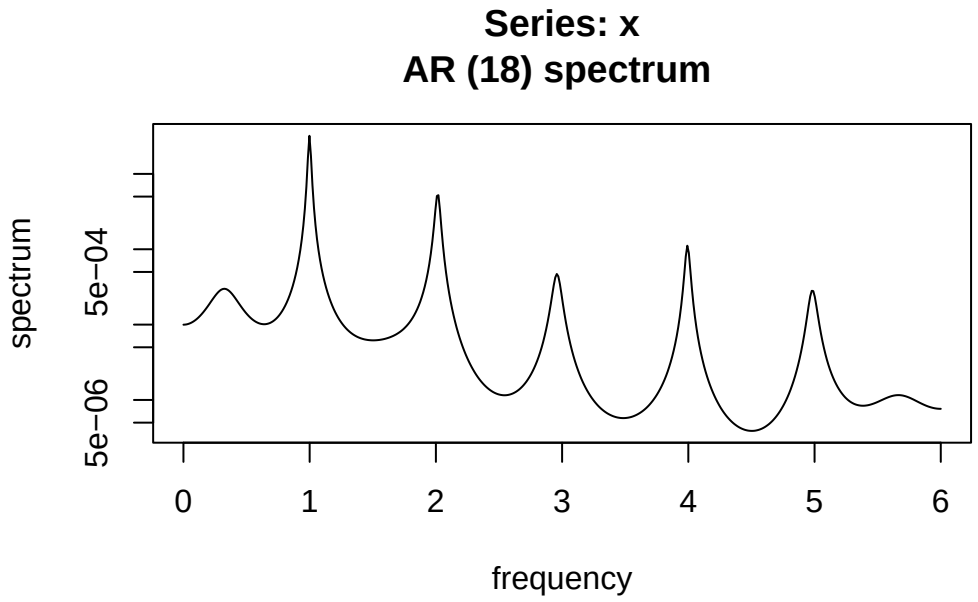


Figure 18.3.: Periodogram of AR process approximating the monthly log-transformed and detrended `AirPassengers` time series.

18.5. Periodogram for unequally-spaced time series

Unequally/unevenly-spaced time series or gaps in observations (missing data) can be handled with the Lomb–Scargle periodogram calculation. This method was originally developed for the analysis of unevenly-spaced astronomical signals (Lomb 1976; Scargle 1982) but found application in many other domains, including environmental science (e.g., Ruf 1999; Campos et al. 2008; Siskey et al. 2016).

Example: Ammonium concentration in wastewater system

For example, consider a time series `imputeTS::tsNH4` of ammonium (NH_4) concentration in a wastewater system (Figure 19.4). This time series contains observations from regular 10-minute

intervals, but some of the observations are missing.

```
forecast::autoplot(imputeTS::tsNH4) +
    xlab("Day") +
    ylab(bquote(NH[4]~concentration))
```

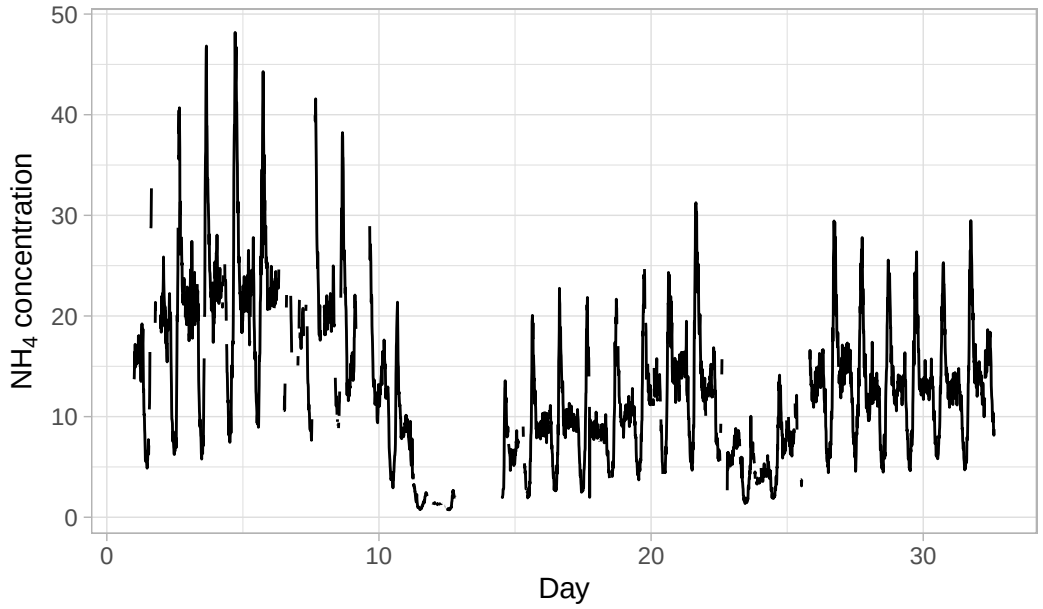


Figure 18.4.: Time series of ammonium concentration in a wastewater system.

Save the data in a format acceptable by the Lomb–Scargle function.

```
D <- data.frame(NH4 = as.vector(imputeTS::tsNH4),
                Time = as.vector(time(imputeTS::tsNH4)))
```

If needed, `na.omit(D)` can be used to remove the rows with missing values.
Figure 19.5 and Figure 19.6 show periodograms calculated based on these data. Note that the function `lomb::lsp()` has arguments adjusting the span of the frequencies and significance level.

```
par(mfrow = c(2, 2))
lomb::lsp(D$NH4, times = D$Time, type = "frequency", alpha = 0.05, main = "")
```

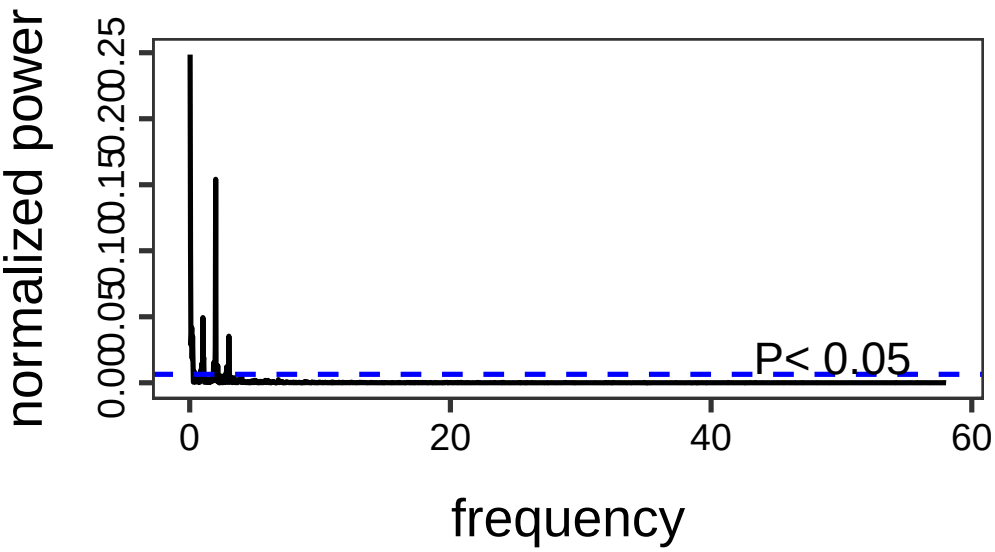


Figure 18.5.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

```
lomb::lsp(D$NH4, times = D$Time, type = "period", alpha = 0.05, main = "")
```

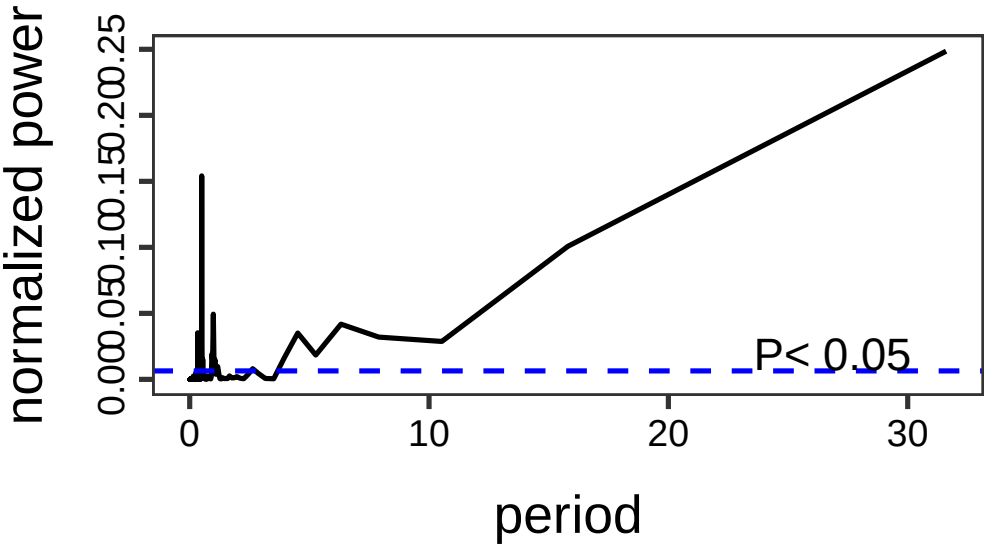


Figure 18.6.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

18.6. Frequency estimation in time

The assumption of a whole time series having a constant spectrum might be very limiting for the analysis of long time series. An estimation of frequencies locally in time allows us to study how the spectrum changes in time, and also detect smaller (short-lived) signals that would be averaged out otherwise. A straightforward solution is to separate the time series into windows (sub-periods) and estimate a spectrum in each such window.

The windowed analysis makes sense when in each window we have enough observations to estimate the frequencies. As a guide, we should have at least two observations per period to be able to represent the signal in the discrete Fourier transform. For example, we cannot estimate the seasonality if we sample once per year – this is our *sampling rate* or *sampling frequency* F_s. To start identifying seasonal periodicity, our sampling rate should be at least 2 samples per year. The highest frequency we can work with is limited by the *Nyquist frequency*

$$F_N = \frac{F_s}{2},$$

which is half of the sampling frequency.

After applying the FFT in each window, the results can be visualized in a *spectrogram*. The spectrogram combines information from multiple periodograms: it shows time on the x-axis, frequency on the y-axis, and the corresponding spectrum power as the color.

Example: Spectrograms for the NAO

Consider a long time series of the North Atlantic Oscillation (NAO) index obtained from an ensemble of 100 model-constrained NAO reconstructions (Figure 19.7).

```
NAOdf <- readr::read_csv("data/NAO.csv", skip = 12, show_col_types = FALSE) %>%
    rename(NAOproxy = nao_mc_mean) %>%
    select(Year, NAOproxy) %>%
    arrange(Year)
NAO <- ts(NAOdf$NAOproxy, start = NAOdf$Year[1])
spec_raw <- spec.pgram(NAO, detrend = FALSE, plot = FALSE, spans = 3)

# Find the frequency with the max power
fmax <- spec_raw$freq[which.max(spec_raw$spec)]
```

```
p1 <- forecast::autoplot(NAO) +
    xlab("Year") +
    ylab("NAO")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p3 <- data.frame(Spectrum = spec_raw$spec,
                 Period = 1/spec_raw$freq) %>%
    ggplot(aes(x = Period, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = 1/fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

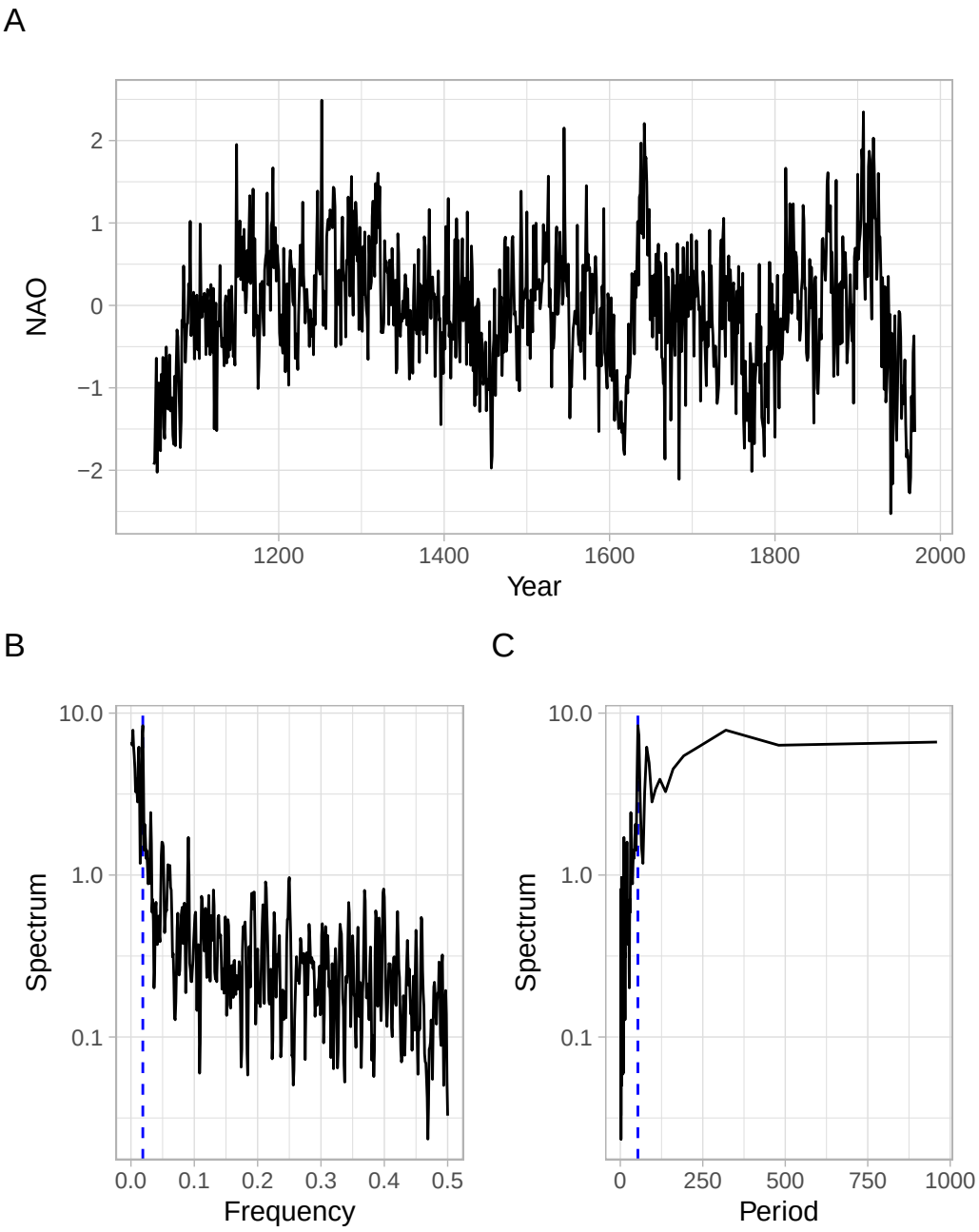


Figure 18.7.: North Atlantic Oscillation (NAO) winter index (December, January, and February) calculated as the mean of 100 model-constrained NAO reconstructions, along with its smoothed periodograms. The dashed lines denote the maximal magnitude of the spectrum.

This is an annual time series, hence the sampling frequency $F_s = 1$. From the global FFT results, the frequency with the maximal power is 0.019 year^{-1}, which is equivalent to the period of about 53.3 years. We will set the window for the FFT and calculate the spectrogram. Note

that we can use overlapping windows for a smoother picture.

```
# Set the window size (years), for applying the FFT in each window
window_size = 30

# Compute the spectrogram
SP <- signal::specgram(x = NAO,
                       n = window_size,
                       overlap = window_size/3,
                       Fs = 1)
```

See the spectrograms in Figure 19.8 and compare them with the time series plot in Figure 19.7 A.

```
par(mfrow = c(1, 2))

# Plot the spectrogram without decorations
print(SP, main = "A) Raw results")

# Discard phase information
P <- abs(SP$S)

# Normalize the spectrum to the range [0, 1]
P <- P - min(P)
P <- P/max(P)

# Extract time
Years <- time(NAO)[SP$t]

# Plot the spectrogram after processing
oce::imagep(x = Years,
            y = SP$f,
            z = t(P),
            col = oce::oce.colorsViridis,
            ylab = expression(Frequency~(y^-1 )),
            xlab = "Year",
            main = "B) After normalization and adding colors",
            drawPalette = TRUE,
            decimate = FALSE)
```

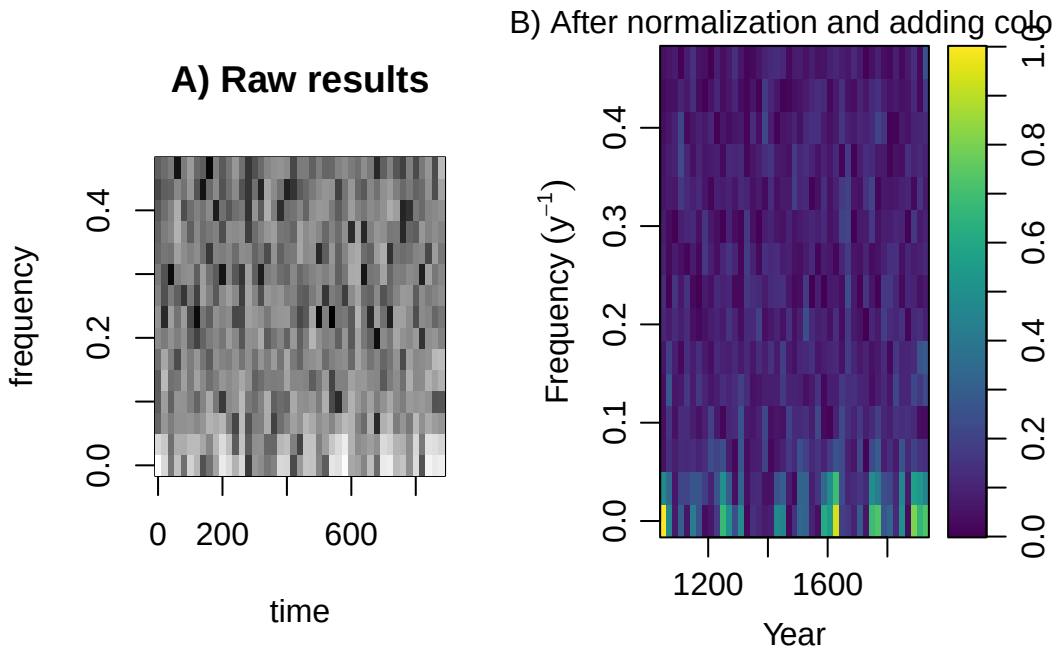


Figure 18.8.: Spectrograms for North Atlantic Oscillation (NAO) winter index.

18.7. Conclusion

For challenging problems, smoothing, multitapering, linear filtering, (repeated) pre-whitening, and Lomb–Scargle can be used together. Beware that aperiodic but autoregressive processes produce peaks in spectral densities. Harmonic analysis is a complicated art rather than a straightforward procedure.

It is extremely difficult to derive the significance of a weak periodicity from harmonic analysis. Do not believe analytical estimates (e.g., exponential probability), as they rarely apply to real data. It is essential to make simulations, typically permuting or bootstrapping the data keeping the observing times fixed. Simulations of the final model with the observation times are also advised.

18.8. Appendix

Wavelet analysis

An alternative to the windowed FFT is the wavelet analysis, which also estimates the strength of time series variations for different frequencies and times. *Wavelet* is a 'small wave' or function $\psi(t)$ approximating the variations. Popular wavelet functions are Meyer, Morlet, and Mexican hat.

Figure 19.9 shows the results of using the Morlet wavelet to process the NAO time series.

```
library(dplR)
wv <- morlet(y1 = NAOdf$NAOproxy, x1 = NAOdf$Year)
levs <- quantile(wv$Power, probs = c(0, 0.5, 0.75, 0.9, 0.95, 1))
wavelet.plot(wv, wavelet.levels = levs)
```

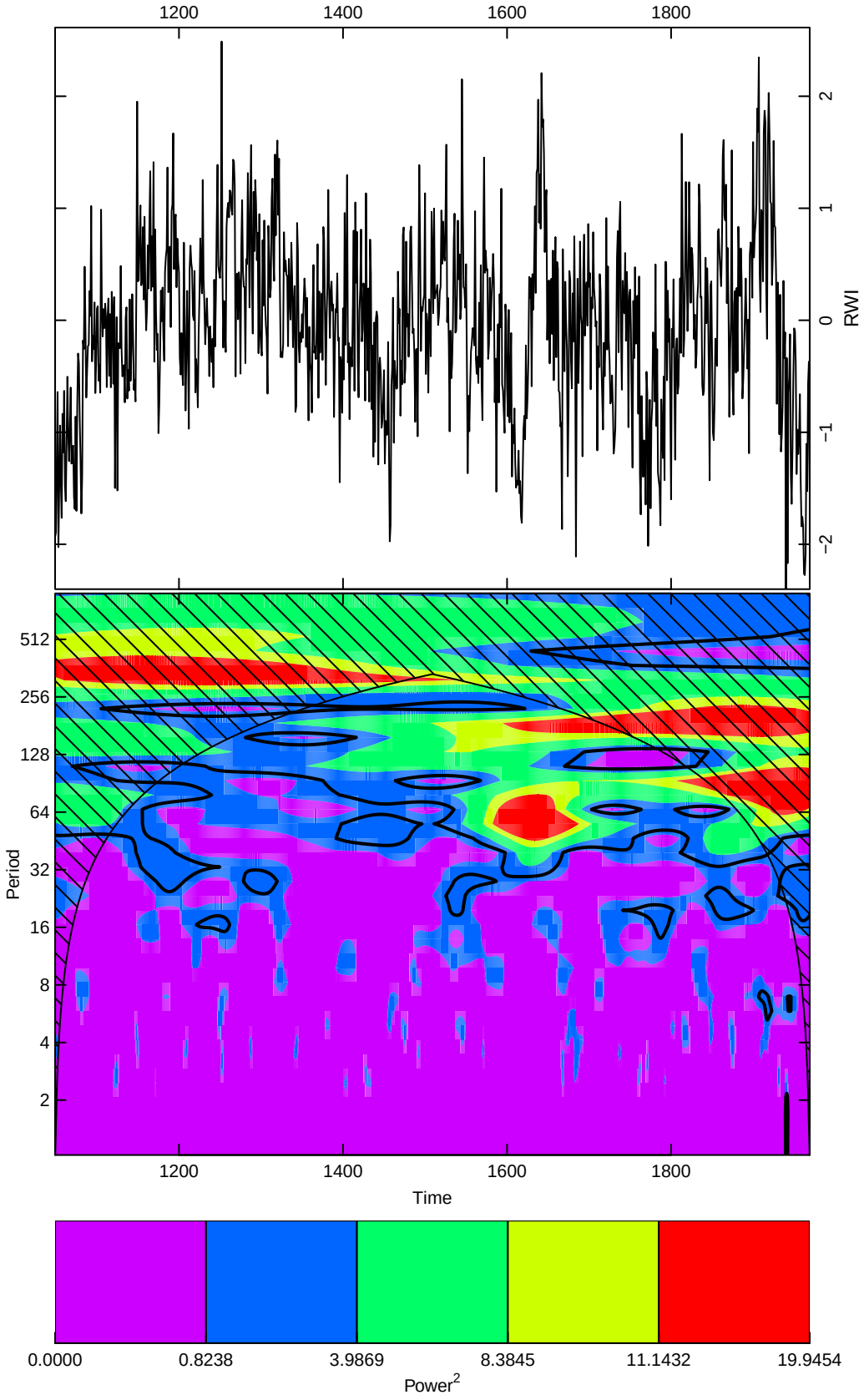


Figure 18.9.: Wavelet analysis of the North Atlantic Oscillation (NAO) winter index.

19. Causality

The goal of this lecture is to learn how to view time series from the frequency perspective. You should be able to recognize features of time series from a periodogram similar to how we did it using plots of the autocorrelation function (ACF).

Objectives

1. Describe the mechanics of Fourier transform for time series.
2. Present results of spectrum calculations using a periodogram or spectrogram.
3. Give examples of smoothed periodograms.
4. Demonstrate the use of the Lomb–Scargle periodogram for analysis of unequally-spaced time series.

Reading materials

- Chapter 4 in Shumway and Stoffer (2017)
- Nason (2008) on wavelet analysis

19.1. Introduction

By now, we have been working in the *time domain*, such that our analysis could be seen as a regression of the present on the past (for example, ARIMA models). We have been using many time series plots with time on the x-axis.

An alternative approach is to analyze time series in a *spectral domain*, such that use a regression of present on a linear combination of sine and cosine functions. In this type of analysis, we often use periodogram plots, with frequency or period on the x-axis.

19.2. Regression on sinusoidal components

The simplest form of spectral analysis consists of regression on a periodic component:

$$Y_t = A\cos\omega t + B\sin\omega t + C + \epsilon_t, \tag{19.1}$$

where $t = 1, \dots, T$ and $\epsilon_t \sim \text{WN}(0, \sigma^2)$. Without loss of generality, we assume $0 \leqslant \omega \leqslant \pi$. In fact, for discrete data, frequencies outside this range are aliased into this range. For example, suppose that $-\pi < (\omega = \pi - \delta) < 0$, then

$$\begin{aligned}\cos(\omega t) &= \cos((\pi - \delta)t) \\ &= \cos(\pi t)\cos(\delta t) + \sin(\pi t)\sin(\delta t) \\ &= \cos(\delta t).\end{aligned}$$

Hence, a sampled sinusoid with a frequency smaller than 0 appears to coincide with a sinusoid with frequency in the interval $[0, \pi]$.

> **Note**
>
> The term $A\cos\omega t + B\sin\omega t$ is a periodic function with the period $2\pi/\omega$. The period $2\pi/\omega$ represents the number of time units that it takes for the function to take the same value again, i.e., to complete a cycle. The frequency, measured in cycles per time unit, is given by the inverse $\omega/(2\pi)$. The angular frequency, measured in radians per time unit, is given by ω. Because of its convenience, the angular frequency ω will be used to describe the periodicity of the function, and its name is shortened to frequency when there is no danger of confusion.

Consider monthly data that exhibit a 12-month seasonality. Hence, the period $2\pi/\omega = 12$, which implies the angular frequency $\omega = \pi/6$. The frequency, measured in cycles per time unit, is given by the inverse

$$\frac{\omega}{2\pi} = \frac{1}{12} \approx 0.08.$$

The formulas to estimate parameters of regression in Equation 19.1 take a much simpler form if ω is one of the Fourier frequencies, defined by

$$\omega_j = \frac{2\pi j}{T}, \quad j = 0, \dots, \frac{T}{2},$$

then

$$\begin{aligned}
\hat{A} &= \frac{2}{T}\sum_t Y_t \cos\omega_j t, \\
\hat{B} &= \frac{2}{T}\sum_t Y_t \sin\omega_j t, \\
\hat{C} &= \overline{Y} = \frac{1}{T}\sum_t Y_t.
\end{aligned}$$

A suitable way of testing the significance of the sinusoidal component with frequency ω_j is using its contribution to the sum of squares

$$R_T(\omega_j) = \frac{T}{2}\left(\hat{A}^2 + \hat{B}^2\right).$$

If the $\epsilon_t \sim N(0, \sigma^2)$, then it follows that $\hat{A}$ and $\hat{B}$ are also independent normal, each with the variance $2\sigma^2/T$, so under the null hypothesis of $A = B = 0$ we find that

$$\frac{R_T(\omega_j)}{\sigma^2} \sim \chi^2_2$$

or equivalently that $R_T(\omega_j)/(2\sigma^2)$ has an exponential distribution with mean 1. The above theory can be extended to the simultaneous estimation of several periodic components.

19.3. Periodogram

The *Fourier transform* uses Fourier series, such as the pairs of sines and cosines with different periods, to describe the frequencies present in the original time series.

The Fourier transform applied to an equally-spaced time series Y_t (where $t = 1, \dots, T$) is also called the *discrete Fourier transform* (DFT) because the time is discrete (not the values Y_t).

To reduce the computational complexity of DFT and speed up the computations, one of the *fast Fourier transform* (FFT) algorithms is typically used.

The results of the Fourier transform are shown in a periodogram that describes the spectral properties of the signal.

The periodogram is defined as

$$I_T(\omega) = \frac{1}{2\pi T}\left|\sum_{t=1}^{T} Y_t e^{i\omega t}\right|^2,$$

which is an approximately unbiased estimator of the spectral density f.

Some undesirable features of the periodogram:

1. $I_T(\omega)$ for fixed ω is not a consistent estimate of $f(\omega)$, since

$$I_T(\omega_j) \sim \frac{f(\omega_j)}{2}\chi_2^2.$$

   Therefore, the variance of $f^2(\omega)$ does not tend to 0 as $T \to \infty$.
2. The independence of periodogram ordinates at different Fourier frequencies suggests that the sample periodogram plotted as a function of ω will be extremely irregular.

Suppose that $\gamma(h)$ is the autocovariance function of a stationary process and that $f(\omega)$ is the spectral density for the same process (h is the time lag and ω is the frequency). The autocovariance $\gamma(h)$ and the spectral density $f(\omega)$ are related:

$$\gamma(h) = \int_{-1/2}^{1/2} e^{2\pi i\omega h} f(\omega) d\omega,$$

and

$$f(\omega) = \sum_{h=-\infty}^{+\infty} \gamma(h) e^{-2\pi i\omega h}.$$

In the language of advanced calculus, the autocovariance and spectral density are Fourier transform pairs. These Fourier transform equations show that there is a direct link between the time domain representation and the frequency domain representation of a time series.

19.4. Smoothing

The idea behind smoothing is to take weighted averages over neighboring frequencies to reduce the variability associated with individual periodogram values. However, such an operation necessarily introduces some bias into the estimation procedure. Theoretical studies focus on the amount of smoothing that is required to obtain an optimum trade-off between bias and variance. In practice, this usually means that the choice of a kernel and amount of smoothing is somewhat subjective.

The main form of a smoothed estimator is given by

$$\hat{f}(\lambda) = \int_{-\pi}^{\pi} \frac{1}{h} K\left(\frac{\omega - \lambda}{h}\right) I_T(\omega) d\omega,$$

where $I_T(\cdot)$ is the periodogram based on T observations, $K(\cdot)$ is a kernel function, and h is the bandwidth. We usually take $K(\cdot)$ to be a nonnegative function, symmetric about 0 and integrating to 1. Thus, any symmetric density, such as the normal density, will work. In practice, however, it is more usual to take a kernel of finite range, such as the *Epanechnikov kernel*

$$K(x) = \frac{3}{4\sqrt{5}}\left(1 - \frac{x^2}{5}\right), \quad -\sqrt{5} \leqslant x \leqslant \sqrt{5},$$

which is 0 outside the interval $[-\sqrt{5}, \sqrt{5}]$. This choice of kernel function has some optimality properties. However, in practice, this optimality is less important than the choice of bandwidth h, which effectively controls the range over which the periodogram is smoothed.

There are some additional difficulties with the performance of the sample periodogram in the presence of a sinusoidal variation whose frequency is not one of the Fourier frequencies. This effect is known as *leakage*. The reason of leakage is that we always consider a truncated periodogram. Truncation implicitly assumes that the time series is periodic with a period T, which, of course, is not always true. So we artificially introduce non-existent periodicities into the estimated spectrum, i.e., cause 'leakage' of the spectrum. (If the time series is perfectly periodic over T then there is no leakage.) The leakage can be treated using an operation of tapering on the periodogram, i.e., by choosing appropriate periodogram windows.

> **Note**
>
> When we work with periodograms, we lose all phase (relative location/time origin) information: the periodogram will be the same if all the data were circularly rotated to a new time origin, i.e., the observed data are treated as perfectly periodic.

Another popular choice to smooth a periodogram is the *Daniell kernel* (with parameter m). For a time series, it is a centered moving average of values between the times $t - m$ and $t + m$ (inclusive). For example, the smoothing formula for a Daniell kernel with $m = 2$ is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + x_{t-1} + x_t + x_{t+1} + x_{t+2}}{5} \\ &= 0.2x_{t-2} + 0.2x_{t-1} + 0.2x_t + 0.2x_{t+1} + 0.2x_{t+2}.\end{aligned}$$

The weighting coefficients for a Daniell kernel with $m = 2$ can be checked using the function `kernel()`. In the output, the indices in `coef[ ]` refer to the time difference from the center of the average at the time t.

```
kernel("daniell", m = 2)
```

```
#> Daniell(2)
#> coef[-2] = 0.2
#> coef[-1] = 0.2
#> coef[ 0] = 0.2
#> coef[ 1] = 0.2
#> coef[ 2] = 0.2
```

The modified Daniell kernel is such that the two endpoints in the averaging receive half the weight that the interior points do. For a modified Daniell kernel with $m = 2$, the smoothing is

$$\begin{aligned}\hat{x}_t &= \frac{x_{t-2} + 2x_{t-1} + 2x_t + 2x_{t+1} + x_{t+2}}{8} \\ &= 0.125x_{t-2} + 0.25x_{t-1} + 0.25x_t + 0.25x_{t+1} + 0.125x_{t+2}\end{aligned}$$

List the weighting coefficients:

```
kernel("modified.daniell", m = 2)
```

```
#> mDaniell(2)
#> coef[-2] = 0.125
#> coef[-1] = 0.250
#> coef[ 0] = 0.250
#> coef[ 1] = 0.250
#> coef[ 2] = 0.125
```

Either the Daniell kernel or the modified Daniell kernel can be convoluted (repeated) so that the smoothing is applied again to the smoothed values. This produces a more extensive smoothing by averaging over a wider time interval. For instance, to repeat a Daniell kernel with $m = 2$ on the smoothed values that resulted from a Daniell kernel with $m = 2$, the formula would be

$$\hat{\hat{x}}_t = \frac{\hat{x}_{t-2} + \hat{x}_{t-1} + \hat{x}_t + \hat{x}_{t+1} + \hat{x}_{t+2}}{5}.$$

The weights for averaging the original data for convoluted Daniell kernels with $m = 2$ in both smooths will be

```
kernel("daniell", m = c(2, 2))
```

```
#> Daniell(2,2)
#> coef[-4] = 0.04
#> coef[-3] = 0.08
#> coef[-2] = 0.12
#> coef[-1] = 0.16
#> coef[ 0] = 0.20
```

```
#> coef[ 1] = 0.16
#> coef[ 2] = 0.12
#> coef[ 3] = 0.08
#> coef[ 4] = 0.04
```

```
kernel("modified.daniell", m = c(2, 2))
```

```
#> mDaniell(2,2)
#> coef[-4] = 0.0156
#> coef[-3] = 0.0625
#> coef[-2] = 0.1250
#> coef[-1] = 0.1875
#> coef[ 0] = 0.2188
#> coef[ 1] = 0.1875
#> coef[ 2] = 0.1250
#> coef[ 3] = 0.0625
#> coef[ 4] = 0.0156
```

The center values are weighted slightly more heavily in the modified Daniell kernel than in the unmodified one.

When we smooth a periodogram, we are smoothing *across frequencies* rather than across times.

Example: Spectral analysis of the airline passenger data

Recall the airline passenger time series (**?@fig-airpassangers**) that has an increasing trend and strong multiplicative seasonality.
The series should be detrended before a spectral analysis. For demonstration purposes, compute periodogram for the raw data and log-transformed and detrended.

```
# Fit a quadratic trend to log-transformed data
t <- as.vector(time(AirPassengers))
mod <- lm(log10(AirPassengers) ~ poly(t, degree = 2))
res <- ts(mod$residuals)
attributes(res) <- attributes(AirPassengers)

# Compute spectra
spec_raw <- spec.pgram(AirPassengers, detrend = FALSE, plot = FALSE)
spec_log_detrend <- spec.pgram(res, detrend = FALSE, plot = FALSE)
```

The x-axes of the periodograms correspond to $\omega/(2\pi)$ or the number of cycles per the time series period specified in the `ts` object (12 months). If the frequency of 12 was not specified for this time series, the x-axes would be different, and the first spectrum peak would correspond to ≈ 0.08.
Figure 19.1 B shows that the spectrum of the raw data is dominated by low frequencies (trend). After detrending the data (Figure 19.1 C), the spectrum at frequency 1 (meaning one cycle per year) is the dominant one.

```
pAirPassengers <- forecast::autoplot(AirPassengers, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers (thousand)") +
    ggtitle("Raw series")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
p3 <- forecast::autoplot(res, col = "grey50") +
    xlab("Year") +
    ylab("Airline passengers residuals (lg[thousand])") +
    ggtitle("Detrended and log-transformed series")
p4 <- data.frame(Spectrum = spec_log_detrend$spec,
                 Frequency = spec_log_detrend$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_line() +
    ggtitle("")
(pAirPassengers + p2) / (p3 + p4) +
    plot_annotation(tag_levels = 'A')
```

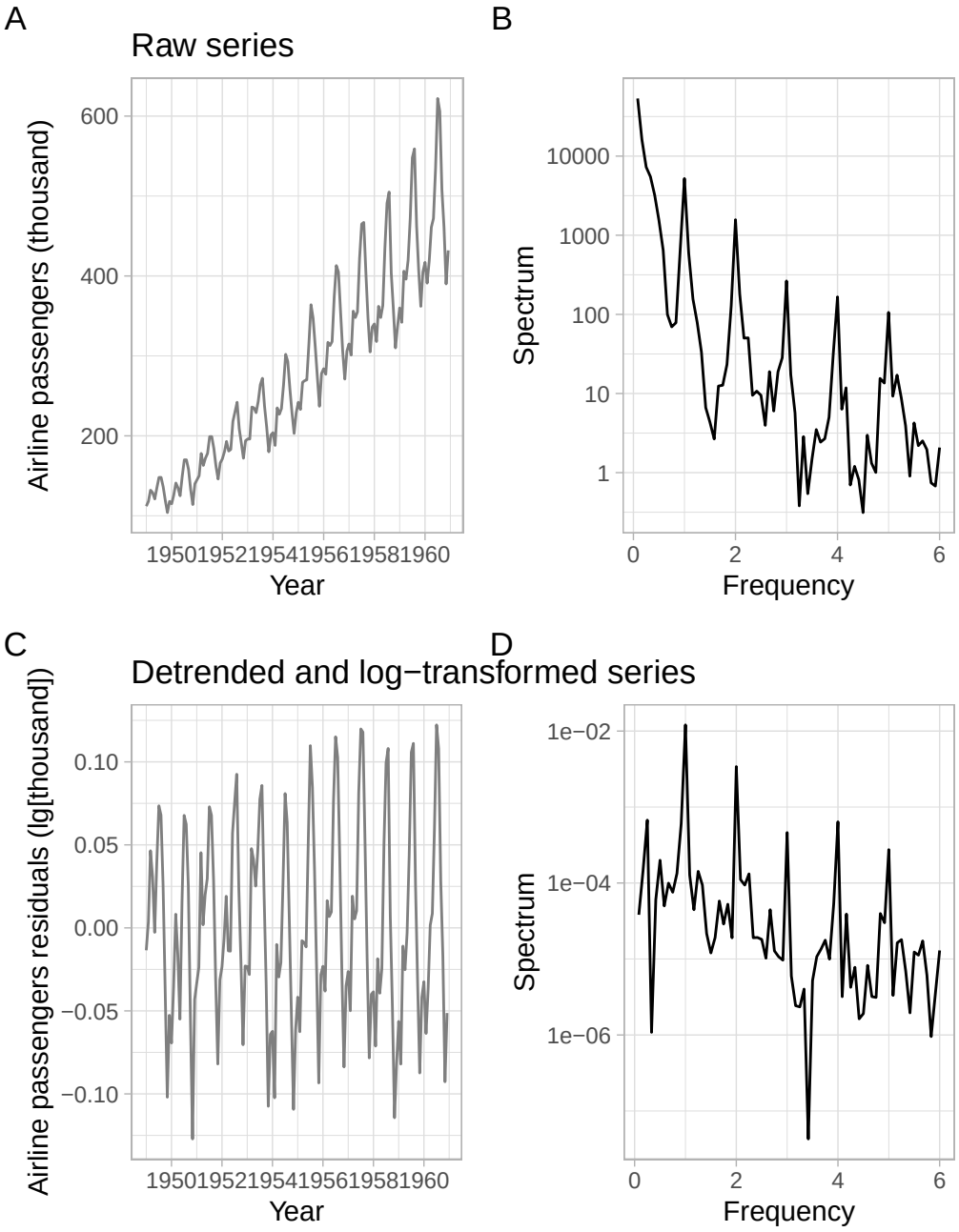


Figure 19.1.: Monthly `AirPassengers` time series, log-transformed and detrended, and the corresponding fast Fourier transforms.

Note that the function `spec.pgram()` has the option of removing a simpler (linear) trend and showing the results as a base-R plot. The confidence band in the upper right corner of this plot helps to identify the statistical significance of the peaks (Figure 19.2).

```
par(mfrow = c(2, 2))
spec.pgram(res, spans = c(3))
spec.pgram(res, spans = c(3, 3))
spec.pgram(res, spans = c(7, 7))
spec.pgram(res, spans = c(11, 11))
```

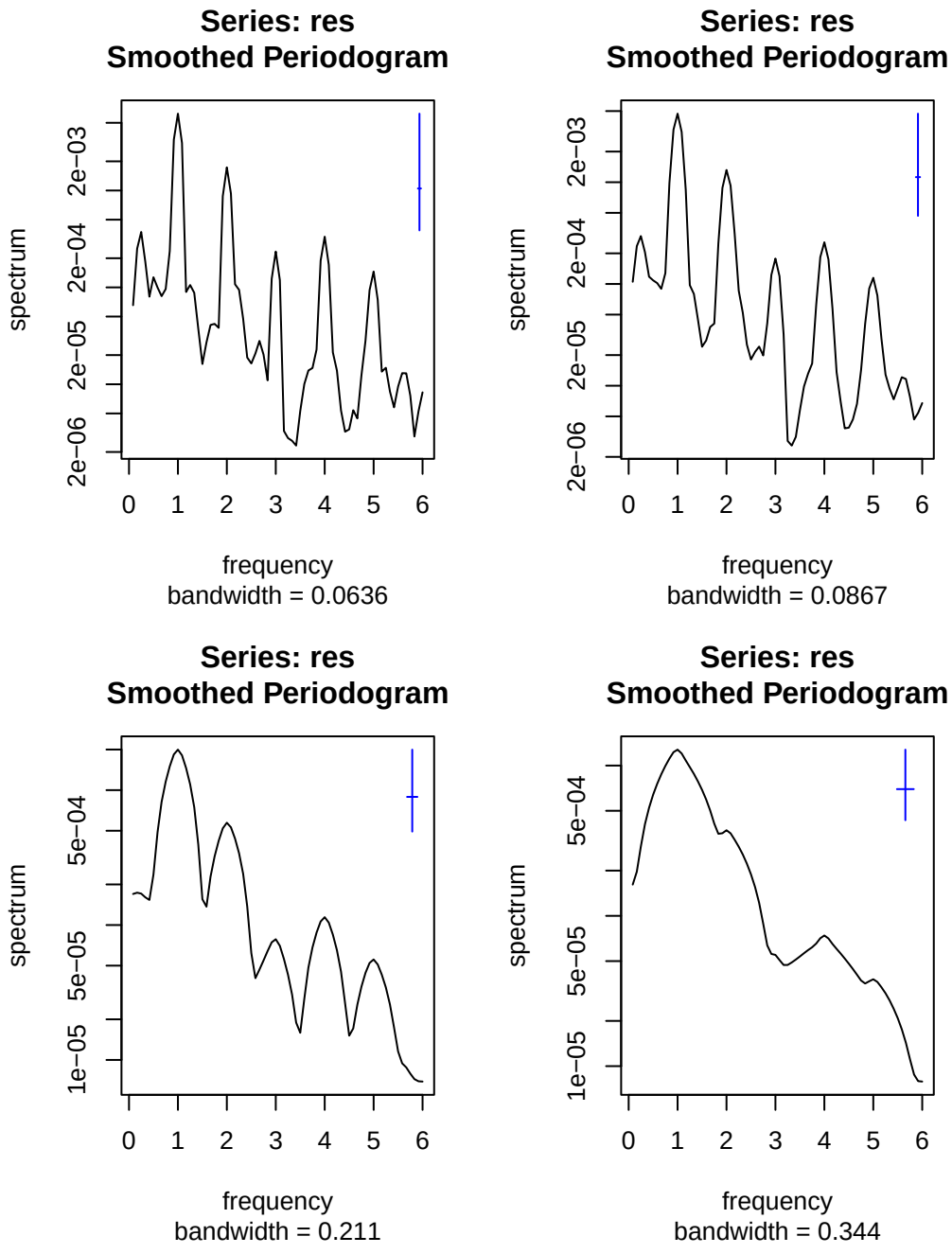


Figure 19.2.: Smoothed periodograms of monthly log-transformed and detrended `AirPassengers` time series.

Another method of testing the reality of a peak is to look at its harmonics. It is extremely unlikely that a true cycle will be shaped perfectly as a sine curve, hence at least a few of the first harmonics will show up as well. For example, in monthly data with annual seasonality (period of 12 months), we often observe peaks at 6, 4, and 3 months. This is exactly the pattern that we see in the figures above.
We can also approximate our data with an AR model and then plot the approximating periodogram of the AR model (Figure 19.3).

```
spectrum(res, method = "ar")
```

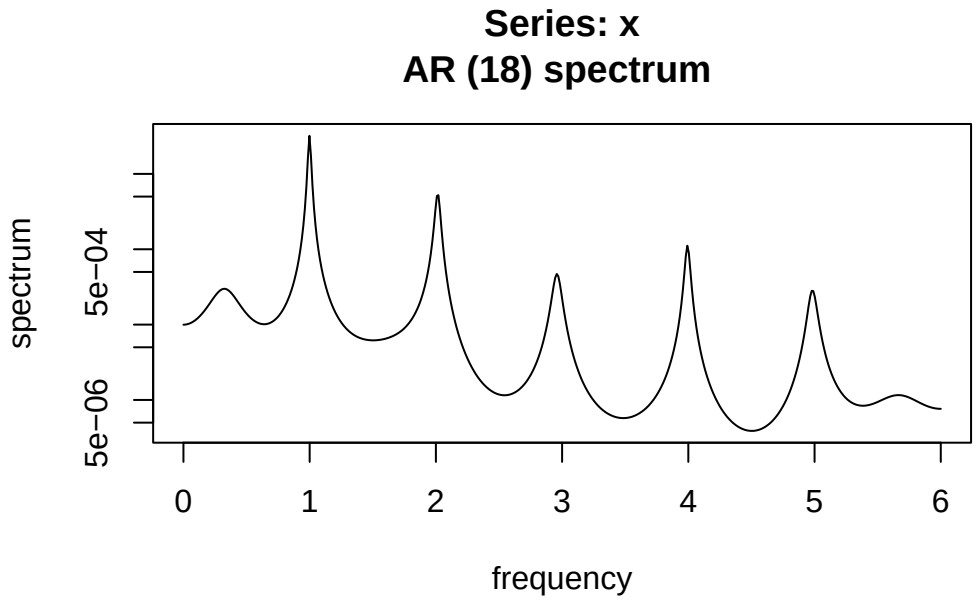


Figure 19.3.: Periodogram of AR process approximating the monthly log-transformed and detrended `AirPassengers` time series.

19.5. Periodogram for unequally-spaced time series

Unequally/unevenly-spaced time series or gaps in observations (missing data) can be handled with the Lomb–Scargle periodogram calculation. This method was originally developed for the analysis of unevenly-spaced astronomical signals (Lomb 1976; Scargle 1982) but found application in many other domains, including environmental science (e.g., Ruf 1999; Campos et al. 2008; Siskey et al. 2016).

Example: Ammonium concentration in wastewater system

For example, consider a time series `imputeTS::tsNH4` of ammonium (NH_4) concentration in a wastewater system (Figure 19.4). This time series contains observations from regular 10-minute

intervals, but some of the observations are missing.

```
forecast::autoplot(imputeTS::tsNH4) +
    xlab("Day") +
    ylab(bquote(NH[4]~concentration))
```

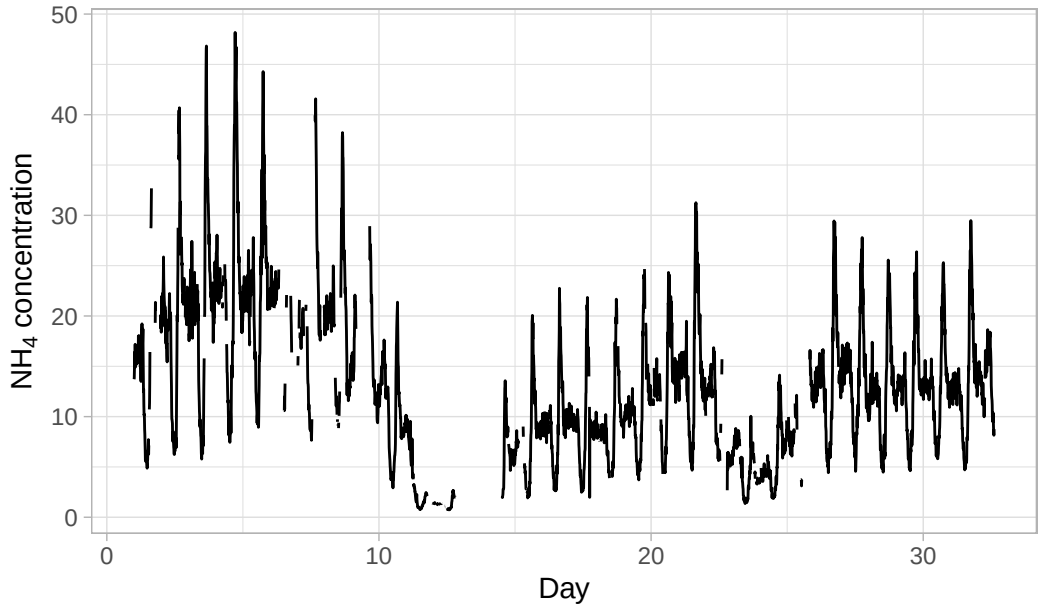


Figure 19.4.: Time series of ammonium concentration in a wastewater system.

Save the data in a format acceptable by the Lomb–Scargle function.

```
D <- data.frame(NH4 = as.vector(imputeTS::tsNH4),
                Time = as.vector(time(imputeTS::tsNH4)))
```

If needed, `na.omit(D)` can be used to remove the rows with missing values.
Figure 19.5 and Figure 19.6 show periodograms calculated based on these data. Note that the function `lomb::lsp()` has arguments adjusting the span of the frequencies and significance level.

```
par(mfrow = c(2, 2))
lomb::lsp(D$NH4, times = D$Time, type = "frequency", alpha = 0.05, main = "")
```

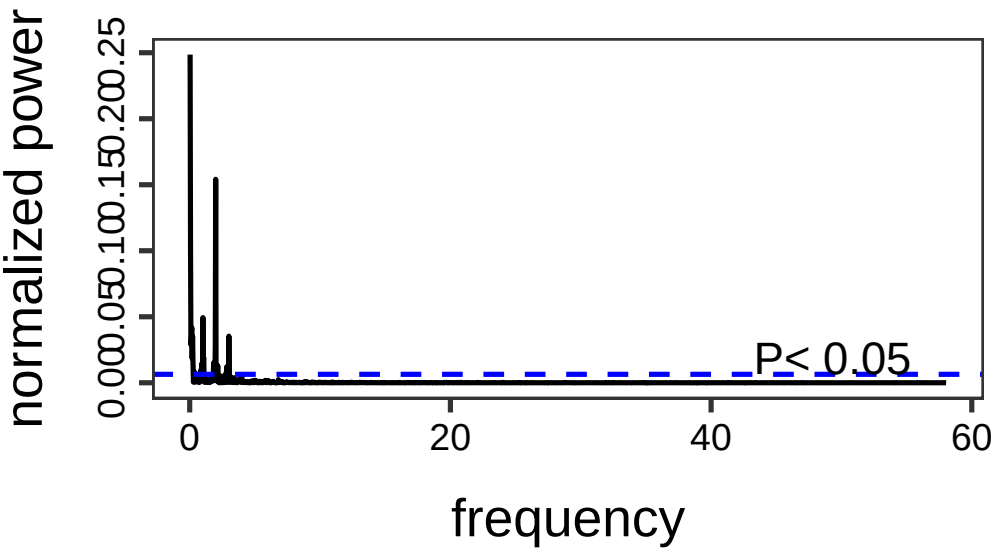


Figure 19.5.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

```
lomb::lsp(D$NH4, times = D$Time, type = "period", alpha = 0.05, main = "")
```

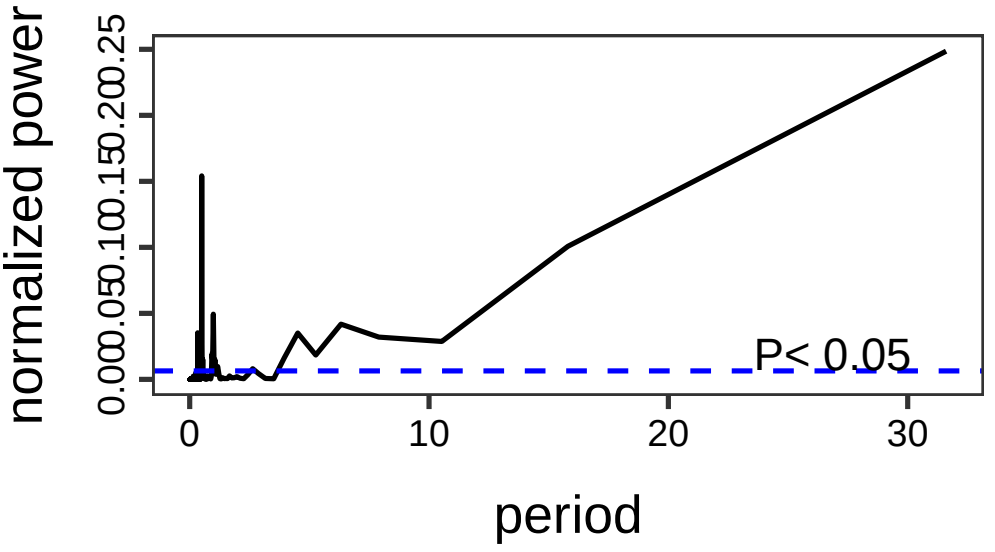


Figure 19.6.: Lomb–Scargle periodograms computed for the unequally-spaced time series of ammonium concentration.

19.6. Frequency estimation in time

The assumption of a whole time series having a constant spectrum might be very limiting for the analysis of long time series. An estimation of frequencies locally in time allows us to study how the spectrum changes in time, and also detect smaller (short-lived) signals that would be averaged out otherwise. A straightforward solution is to separate the time series into windows (sub-periods) and estimate a spectrum in each such window.

The windowed analysis makes sense when in each window we have enough observations to estimate the frequencies. As a guide, we should have at least two observations per period to be able to represent the signal in the discrete Fourier transform. For example, we cannot estimate the seasonality if we sample once per year – this is our *sampling rate* or *sampling frequency* F_s. To start identifying seasonal periodicity, our sampling rate should be at least 2 samples per year. The highest frequency we can work with is limited by the *Nyquist frequency*

$$F_N = \frac{F_s}{2},$$

which is half of the sampling frequency.

After applying the FFT in each window, the results can be visualized in a *spectrogram*. The spectrogram combines information from multiple periodograms: it shows time on the x-axis, frequency on the y-axis, and the corresponding spectrum power as the color.

Example: Spectrograms for the NAO

Consider a long time series of the North Atlantic Oscillation (NAO) index obtained from an ensemble of 100 model-constrained NAO reconstructions (Figure 19.7).

```
NAOdf <- readr::read_csv("data/NAO.csv", skip = 12, show_col_types = FALSE) %>%
    rename(NAOproxy = nao_mc_mean) %>%
    select(Year, NAOproxy) %>%
    arrange(Year)
NAO <- ts(NAOdf$NAOproxy, start = NAOdf$Year[1])
spec_raw <- spec.pgram(NAO, detrend = FALSE, plot = FALSE, spans = 3)

# Find the frequency with the max power
fmax <- spec_raw$freq[which.max(spec_raw$spec)]
```

```
p1 <- forecast::autoplot(NAO) +
    xlab("Year") +
    ylab("NAO")
p2 <- data.frame(Spectrum = spec_raw$spec,
                 Frequency = spec_raw$freq) %>%
    ggplot(aes(x = Frequency, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p3 <- data.frame(Spectrum = spec_raw$spec,
                 Period = 1/spec_raw$freq) %>%
    ggplot(aes(x = Period, y = Spectrum)) +
    scale_y_continuous(trans = 'log10') +
    geom_vline(xintercept = 1/fmax, col = "blue", lty = 2) +
    geom_line() +
    ggtitle("")
p1 / (p2 + p3) +
    plot_annotation(tag_levels = 'A')
```

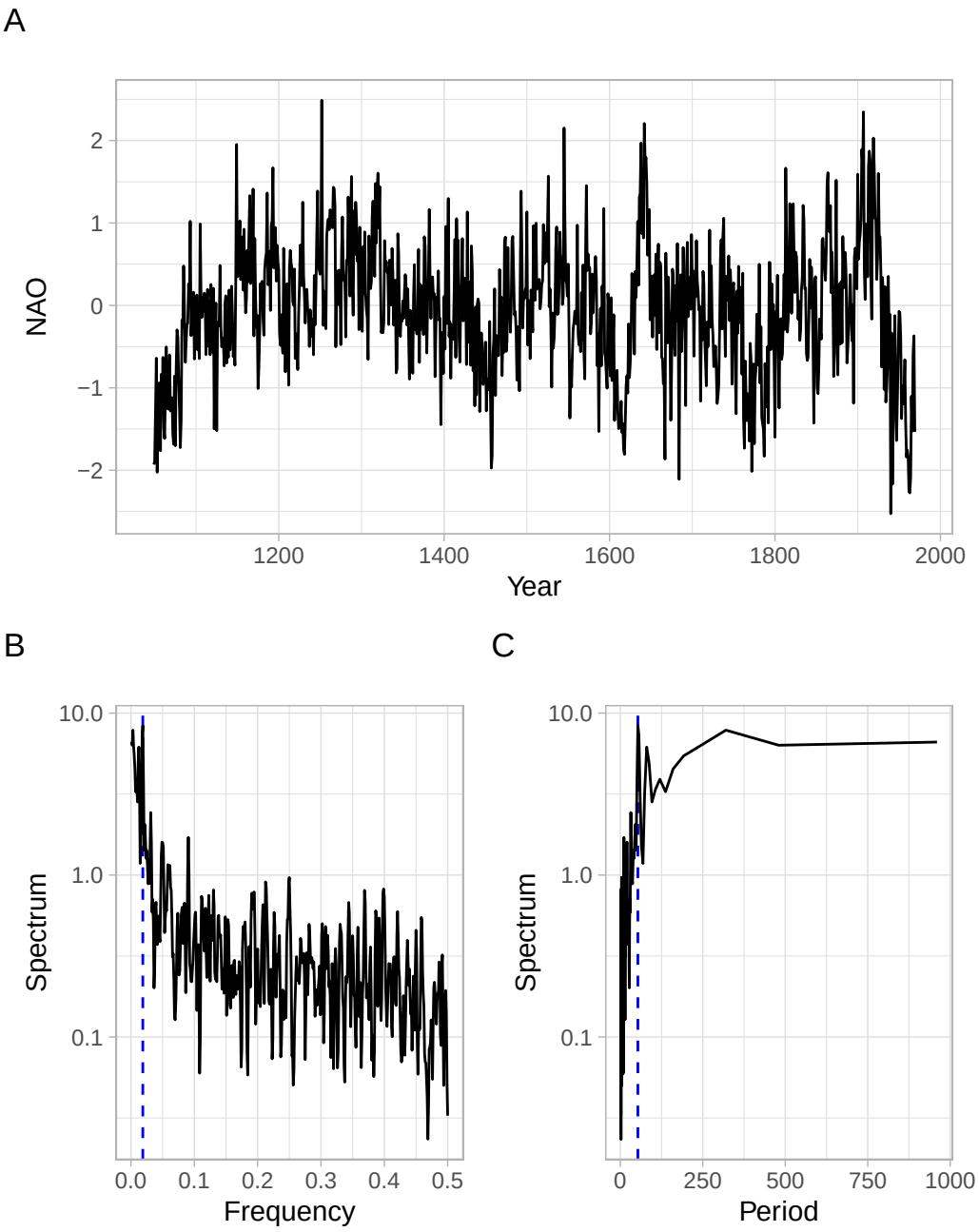


Figure 19.7.: North Atlantic Oscillation (NAO) winter index (December, January, and February) calculated as the mean of 100 model-constrained NAO reconstructions, along with its smoothed periodograms. The dashed lines denote the maximal magnitude of the spectrum.

This is an annual time series, hence the sampling frequency $F_s = 1$. From the global FFT results, the frequency with the maximal power is 0.019 year^{-1}, which is equivalent to the period of about 53.3 years. We will set the window for the FFT and calculate the spectrogram. Note

that we can use overlapping windows for a smoother picture.

```
# Set the window size (years), for applying the FFT in each window
window_size = 30

# Compute the spectrogram
SP <- signal::specgram(x = NAO,
                       n = window_size,
                       overlap = window_size/3,
                       Fs = 1)
```

See the spectrograms in Figure 19.8 and compare them with the time series plot in Figure 19.7 A.

```
par(mfrow = c(1, 2))

# Plot the spectrogram without decorations
print(SP, main = "A) Raw results")

# Discard phase information
P <- abs(SP$S)

# Normalize the spectrum to the range [0, 1]
P <- P - min(P)
P <- P/max(P)

# Extract time
Years <- time(NAO)[SP$t]

# Plot the spectrogram after processing
oce::imagep(x = Years,
            y = SP$f,
            z = t(P),
            col = oce::oce.colorsViridis,
            ylab = expression(Frequency~(y^-1 )),
            xlab = "Year",
            main = "B) After normalization and adding colors",
            drawPalette = TRUE,
            decimate = FALSE)
```

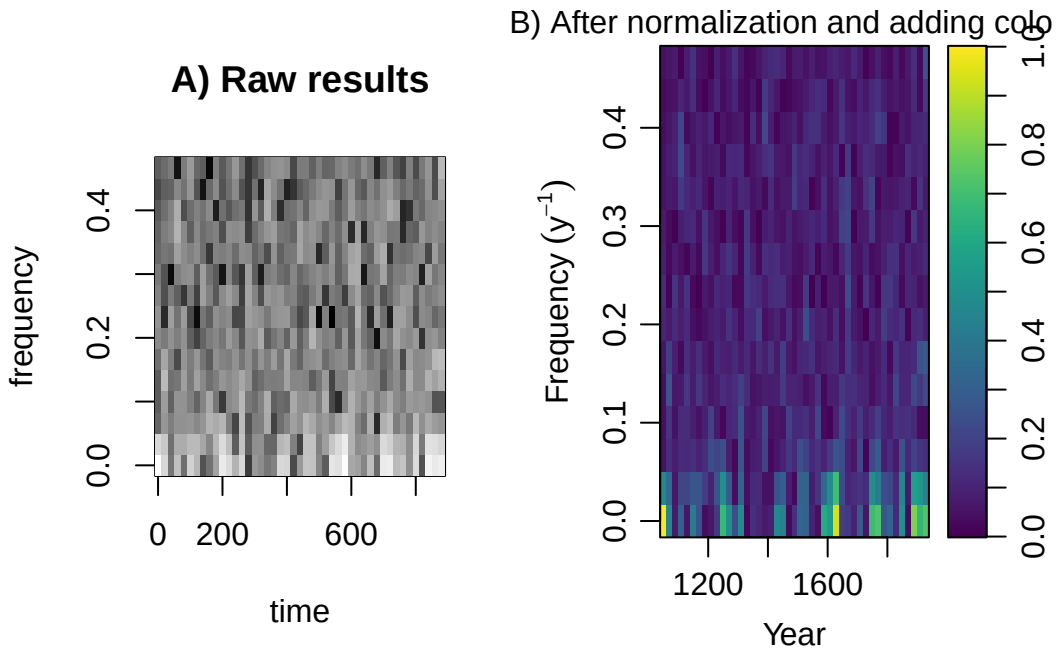


Figure 19.8.: Spectrograms for North Atlantic Oscillation (NAO) winter index.

19.7. Conclusion

For challenging problems, smoothing, multitapering, linear filtering, (repeated) pre-whitening, and Lomb–Scargle can be used together. Beware that aperiodic but autoregressive processes produce peaks in spectral densities. Harmonic analysis is a complicated art rather than a straightforward procedure.

It is extremely difficult to derive the significance of a weak periodicity from harmonic analysis. Do not believe analytical estimates (e.g., exponential probability), as they rarely apply to real data. It is essential to make simulations, typically permuting or bootstrapping the data keeping the observing times fixed. Simulations of the final model with the observation times are also advised.

19.8. Appendix

Wavelet analysis

An alternative to the windowed FFT is the wavelet analysis, which also estimates the strength of time series variations for different frequencies and times. *Wavelet* is a 'small wave' or function $\psi(t)$ approximating the variations. Popular wavelet functions are Meyer, Morlet, and Mexican hat.

Figure 19.9 shows the results of using the Morlet wavelet to process the NAO time series.

```
library(dplR)
wv <- morlet(y1 = NAOdf$NAOproxy, x1 = NAOdf$Year)
levs <- quantile(wv$Power, probs = c(0, 0.5, 0.75, 0.9, 0.95, 1))
wavelet.plot(wv, wavelet.levels = levs)
```

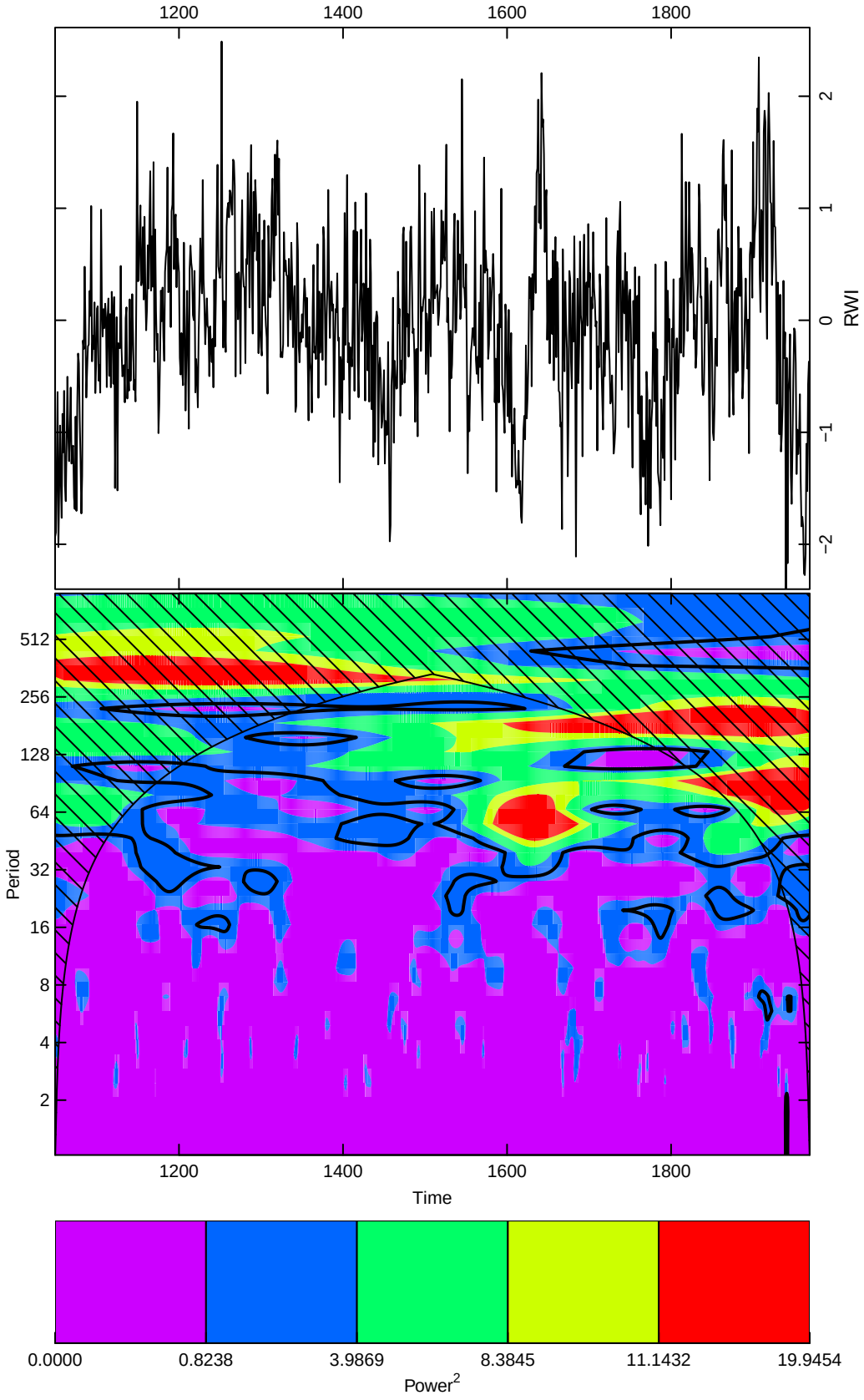


Figure 19.9.: Wavelet analysis of the North Atlantic Oscillation (NAO) winter index.

References

Allaire J, Xie Y, Dervieux C, et al (2026) Rmarkdown: Dynamic documents for r. R package version 2.31, https://github.com/rstudio/rmarkdown

Benson B, Magnuson J, Sharma S (2020) Global lake and river ice phenology database. Version 1 (G01377). National Snow and Ice Data Center, Boulder, CO, USA

Bickel PJ, Götze F, Zwet WR van (1997) Resampling fewer than n observations: Gains, losses, and remedies for losses. Statistica Sinica 7:1–31

Bollerslev T (1986) Generalized autoregressive conditional heteroskedasticity. Journal of Econometrics 31:307–327. https://doi.org/10.1016/0304-4076(86)90063-1

Bollerslev T (2009) Glossary to ARCH (GARCH). In: Volatility and time series econometrics: Essays in honour of Robert F. Engle. SSRN

Borchers HW (2025) Pracma: Practical numerical math functions. R package version 2.4.6, https://CRAN.R-project.org/package=pracma

Box GEP, Jenkins GM (1976) Time series analysis: Forecasting and control. Holden-Day, San Francisco, CA, USA

Brockwell PJ, Davis RA (2002) Introduction to time series and forecasting, 2nd edn. Springer, New York, NY, USA

Brooks C, Burke SP (2003) Information criteria for GARCH model selection. The European Journal of Finance 9:557–580. https://doi.org/10.1080/1351847021000029188

Bühlmann P (2002) Bootstraps for time series. Statistical Science 17:52–72. https://doi.org/10.1214/ss/1023798998

Bunn A, Korpela M, Biondi F, et al (2025) dplR: Dendrochronology program library in r. R package version 1.7.8, https://github.com/OpenDendro/dplR

Cabilio P, Zhang Y, Chen X (2013) Bootstrap rank tests for trend in time series. Environmetrics 24:537–549. https://doi.org/10.1002/env.2250

Caeiro F, Mateus A (2024) Randtests: Testing randomness in r. R package version 1.0.2, https://CRAN.R-project.org/package=randtests

Campbell SD, Diebold FX (2005) Weather forecasting for weather derivatives. Journal of the

American Statistical Association 100:6–16. https://doi.org/10.1198/016214504000001051

Campos MC, Costa JL, Quintella BR, et al (2008) Activity and movement patterns of the Lusitanian toadfish inferred from pressure-sensitive data-loggers in the Mira estuary (Portugal). Fisheries Management and Ecology 15:449–458. https://doi.org/10.1111/j.1365-2400.2008.00629.x

Chatfield C (2000) Time-series forecasting. CRC Press, Boca Raton, FL, USA

Chatterjee S, Hadi AS (2006) Regression analysis by example, 4th edn. John Wiley & Sons, Hoboken, NJ, USA

Chatterjee S, Simonoff JS (2013) Handbook of regression analysis. John Wiley & Sons, Hoboken, NJ, USA

Cochrane D, Orcutt GH (1949) Application of least squares regression to relationships containing auto-correlated error terms. Journal of the American Statistical Association 44:32–61. https://doi.org/10.2307/2280349

Cripps E, Dunsmuir WTM (2003) Modeling the variability of Sydney Harbor wind measurements. Journal of Applied Meteorology 42:1131–1138. https://doi.org/10.1175/1520-0450(2003)042%3C1131:MTVOSH%3E2.0.CO;2

Croissant Y, Graves S (2025) Ecdat: Data sets for econometrics. R package version 0.4.7, https://www.r-project.org

Davison AC, Hinkley DV (1997) Bootstrap methods and their application. Cambridge University Press, Cambridge, UK

Degras D, Xu Z, Zhang T, Wu WB (2012) Testing for parallelism among trends in multiple time series. IEEE Transactions on Signal Processing 60:1087–1097. https://doi.org/10.1109/TSP.2011.2177831

Dickey DA, Fuller WA (1979) Distribution of the estimators for autoregressive time series with a unit root. Journal of the American Statistical Association 74:427–431. https://doi.org/10.2307/2286348

Duguay CR, Brown L, Kang K-K, Kheyrollah Pour H (2013) State of the climate in 2012: Lake ice. Bulletin of the American Meteorological Society 94:S124–S126. https://doi.org/10.1175/2013BAMSStateoftheClimate.1

Efron B (1979) Bootstrap methods: Another look at the jackknife. The Annals of Statistics 7:1–26. https://doi.org/10.1214/aos/1176344552

Engle RF (1982) Autoregressive conditional heteroscedasticity with estimates of the variance of United Kingdom inflation. Econometrica 50:987–1007. https://doi.org/10.2307/1912773

Engle RF, Granger CWJ (1987) Co-integration and error correction: Representation, estimation, and testing. Econometrica 55:251–276. https://doi.org/10.2307/1913236

Esterby SR (1996) Review of methods for the detection and estimation of trends with emphasis on water quality applications. Hydrological Processes 10:127–149. https://doi.org/10.1002/(SICI)1099-1085(199602)10:2%3C127::AID-HYP354%3E3.0.CO;2-8

Eun CS, Lee J (2010) Mean-variance convergence around the world. Journal of Banking & Finance 34:856–870. https://doi.org/10.1016/j.jbankfin.2009.09.016

Fasiolo M, Nedellec R (2025) mgcViz: Visualisations for generalized additive models. R package version 0.2.1, https://github.com/mfasiolo/mgcViz

Gastwirth JL, Gel YR, Hui WLW, et al (2023) Lawstat: Tools for biostatistics, public policy, and law. R package version 3.6, https://CRAN.R-project.org/package=lawstat

Graves S (2024) FinTS: Companion to tsay (2005) analysis of financial time series. R package version 0.4-9, https://geobosh.github.io/FinTSDoc/

Hansen PR, Lunde A (2005) A comparison of volatility models: Does anything beat a GARCH(1, 1)? Journal of Applied Econometrics 20:873–889. https://doi.org/10.1002/jae.800

Härdle W, Horowitz J, Kreiss J-P (2003) Bootstrap methods for time series. International Statistical Review 71:435–459. https://doi.org/10.1111/j.1751-5823.2003.tb00485.x

Hastie TJ, Tibshirani RJ, Friedman JH (2009) The elements of statistical learning: Data mining, inference, and prediction, 2nd edn. Springer, New York, NY, USA

Hirsch RM, Slack JR, Smith RA (1982) Techniques of trend analysis for monthly water quality data. Water Resources Research 18:107–121. https://doi.org/10.1029/WR018i001p00107

Hothorn T, Zeileis A, Farebrother RW, Cummins C (2022) Lmtest: Testing linear regression models. R package version 0.9-40, https://CRAN.R-project.org/package=lmtest

Hyndman R (2023) Fma: Data sets from "forecasting: Methods and applications" by makridakis, wheelwright & hyndman (1998). R package version 2.5, https://pkg.robjhyndman.com/fma/

Hyndman R, Athanasopoulos G, Bergmeir C, et al (2026) Forecast: Forecasting functions for time series and linear models. R package version 9.0.2, https://pkg.robjhyndman.com/forecast/

Kassambara A (2026) Ggpubr: ggplot2 based publication ready plots. R package version 0.6.3, https://rpkgs.datanovia.com/ggpubr/

Kelley D, Richards C (2024) Oce: Analysis of oceanographic data. R package version 1.8-3, https://dankelley.github.io/oce/

Kendall MG (1975) Rank correlation methods, 4th edn. Charles Griffin, London, UK

Kirchgässner G, Wolters J (2007) Introduction to modern time series analysis. Springer-Verlag, Berlin, Germany

Kreiss J-P, Paparoditis E, Politis DN (2011) On the range of validity of the autoregressive sieve bootstrap. Annals of Statistics 39:2103–2130. https://doi.org/10.1214/11-AOS900

Krispin R (2023) TSstudio: Functions for time series analysis and forecasting. R package version 0.1.7, https://github.com/RamiKrispin/TSstudio

Latifovic R, Pouliot D (2007) Analysis of climate change impacts on lake ice phenology in Canada using the historical satellite data record. Remote Sensing of Environment 106:492–507. https://doi.org/10.1016/j.rse.2006.09.015

Ligges U, Short T, Kienzle P (2024) Signal: Signal processing. R package version 1.8-1, https://signal.R-forge.R-project.org

Lomb NR (1976) Least-squares frequency analysis of unequally spaced data. Astrophysics and Space Science 39:447–462. https://doi.org/10.1007/BF00648343

Lyubchich V (2016) Detecting time series trends and their synchronization in climate data. Intelligence Innovations Investments 12:132–137. https://www.researchgate.net/publication/318283780_Detecting_time_series_trends_and_their_synchronization_in_climate_data

Lyubchich V, Gel YR (2016) A local factor nonparametric test for trend synchronism in multiple time series. Journal of Multivariate Analysis 150:91–104. https://doi.org/10.1016/j.jmva.2016.05.004

Lyubchich V, Gel YR, El-Shaarawi A (2013) On detecting non-monotonic trends in environmental time series: A fusion of local regression and bootstrap. Environmetrics 24:209–226. https://doi.org/10.1002/env.2212

Lyubchich V, Gel YR, Vishwakarma S (2025) Funtimes: Functions for time series analysis. R package version 10.0, https://CRAN.R-project.org/package=funtimes

Lyubchich V, Wang X, Heyes A, Gel YR (2016) A distribution-free m-out-of-n bootstrap approach to testing symmetry about an unknown median. Computational Statistics & Data Analysis 104:1–9. https://doi.org/10.1016/j.csda.2016.05.004

Marinova D, McAleer M (2003) Modelling trends and volatility in ecological patents in the USA. Environmental Modelling & Software 18:195–203. https://doi.org/10.1016/S1364-8152(02)00079-8

McLeod AI (2025) Kendall: Kendall rank correlation and mann-kendall trend test. R package version 2.2.2, http://www.stats.uwo.ca/faculty/aim

Nason GP (2008) Wavelet methods in statistics with R. Springer, New York, NY, USA

Noguchi K, Gel YR, Duguay CR (2011) Bootstrap-based tests for trends in hydrological time series, with application to ice phenology data. Journal of Hydrology 410:150–161. https://doi.org/10.1016/j.jhydrol.2011.09.008

O'Hara-Wild M, Hyndman R, Wang E (2026a) Fable: Forecasting models for tidy time series. R

package version 0.5.0, https://fable.tidyverts.org

O'Hara-Wild M, Hyndman R, Wang E (2026b) Feasts: Feature extraction and statistics for time series. R package version 0.5.0, http://feasts.tidyverts.org/

Park C, Hannig J, Kang K-H (2014) Nonparametric comparison of multiple regression curves in scale-space. Journal of Computational and Graphical Statistics 23:657–677. https://doi.org/10.1080/10618600.2013.822816

Park C, Vaughan A, Hannig J, Kang K-H (2009) SiZer analysis for the comparison of time series. Journal of Statistical Planning and Inference 139:3974–3988. https://doi.org/10.1016/j.jspi.2009.05.003

Pedersen TL (2025) Patchwork: The composer of plots. R package version 1.3.2, https://patchwork.data-imaginist.com

Pfaff B (2024) Urca: Unit root and cointegration tests for time series data. R package version 1.3-4, https://CRAN.R-project.org/package=urca

Pinheiro J, Bates D, R Core Team (2025) Nlme: Linear and nonlinear mixed effects models. R package version 3.1-168, https://svn.r-project.org/R-packages/trunk/nlme/

Powell AM, Xu J (2011) Abrupt climate regime shifts, their potential forcing and fisheries impacts. Atmospheric and Climate Sciences 1:33. https://doi.org/10.4236/acs.2011.12004

Rice J (1984) Bandwidth choice for nonparametric regression. The Annals of Statistics 12:1215–1230. https://doi.org/10.1214/aos/1176346788

Ruf T (1999) The Lomb–Scargle periodogram in biological rhythm research: Analysis of incomplete and unequally spaced time-series. Biological Rhythm Research 30:178–201. https://doi.org/10.1076/brhm.30.2.178.1422

Ruf T (2024) Lomb: Lomb-scargle periodogram. R package version 2.5.0, https://CRAN.R-project.org/package=lomb

Rydberg TH (2000) Realistic statistical modelling of financial data. International Statistical Review 68:233–258. https://doi.org/10.2307/1403412

Scargle JD (1982) Studies in astronomical time series analysis. II – statistical aspects of spectral analysis of unevenly spaced data. Astrophysical Journal 263:835–853. https://doi.org/10.1086/160554

Schloerke B, Cook D, Larmarange J, et al (2025) GGally: Extension to ggplot2. R package version 2.4.0, https://ggobi.github.io/ggally/

Seidel DJ, Lanzante JR (2004) An assessment of three alternatives to linear trends for characterizing global atmospheric temperature changes. Journal of Geophysical Research: Atmospheres 109: https://doi.org/10.1029/2003JD004414

Shumway RH, Stoffer DS (2017) Time series analysis and its applications with R examples, 4th edn. Springer, New York, NY, USA

Sievert C, Parmer C, Hocking T, et al (2026) Plotly: Create interactive web graphics via plotly.js. R package version 4.12.0, https://plotly-r.com

Siskey MR, Lyubchich V, Liang D, et al (2016) Periodicity of strontium:calcium across annuli further validates otolith-ageing for Atlantic bluefin tuna (Thunnus thynnus). Fisheries Research 177:13–17. https://doi.org/10.1016/j.fishres.2016.01.004

Stasinopoulos M, Rigby R (2025) Gamlss: Generalized additive models for location scale and shape. R package version 5.5-0, https://www.gamlss.com/

Stoffer D (2026) Astsa: Applied statistical time series analysis. R package version 2.5, https://dsstoffer.github.io/

Taylor JW, Buizza R (2004) A comparison of temperature density forecasts from GARCH and atmospheric models. Journal of Forecasting 23:337–355. https://doi.org/10.1002/for.917

Trapletti A, Hornik K (2026) Tseries: Time series analysis and computational finance. R package version 0.10-61, https://CRAN.R-project.org/package=tseries

Tsay RS (2005) Analysis of financial time series, 2nd edn. John Wiley & Sons, Hoboken, NJ, USA

Vilar-Fernández JM, González-Manteiga W (2004) Nonparametric comparison of curves with dependent errors. Statistics 38:81–99. https://doi.org/10.1080/02331880310001634656

Vogelsang TJ, Franses PH (2005) Testing for common deterministic trend slopes. Journal of Econometrics 126:1–24. https://doi.org/10.1016/j.jeconom.2004.02.004

Wang L, Akritas MG, Van Keilegom I (2008) An ANOVA-type nonparametric diagnostic test for heteroscedastic regression models. Journal of Nonparametric Statistics 20:365–382. https://doi.org/10.1080/10485250802066112

Wickham H (2025) Downlit: Syntax highlighting and automatic linking. R package version 0.4.5, https://downlit.r-lib.org/

Wickham H, Chang W, Henry L, et al (2026a) ggplot2: Create elegant data visualisations using the grammar of graphics. R package version 4.0.3, https://ggplot2.tidyverse.org

Wickham H, François R, Henry L, et al (2026b) Dplyr: A grammar of data manipulation. R package version 1.2.1, https://dplyr.tidyverse.org

Wickham H, Hester J, Bryan J (2026c) Readr: Read rectangular text data. R package version 2.2.0, https://readr.tidyverse.org

Wickham H, Hester J, Ooms J (2026d) xml2: Parse XML. R package version 1.5.2, https://xml2.r-lib.org

Wood S (2025) Mgcv: Mixed GAM computation vehicle with automatic smoothness estimation. R package version 1.9-4, https://CRAN.R-project.org/package=mgcv

Wooldridge JM (2013) Introductory econometrics: A modern approach, 5th edn. Cengage Learning, Mason, OH, USA

Wuertz D, Chalabi Y, Setz T, et al (2025) fGarch: Rmetrics - autoregressive conditional heteroskedastic modelling. R package version 4052.93, https://geobosh.github.io/fGarchDoc/

Xie Y (2025) Knitr: A general-purpose package for dynamic report generation in r. R package version 1.51, https://yihui.org/knitr/

Zeileis A (2019) Dynlm: Dynamic linear regression. R package version 0.3-6, https://CRAN.R-project.org/package=dynlm

Zhang T (2013) Clustering high-dimensional time series based on parallelism. Journal of the American Statistical Association 108:577–588. https://doi.org/10.1080/01621459.2012.760458

A. Weighted least squares

We can often hypothesize that the standard deviation of residuals in the model

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \tag{A.1}$$

is proportional to the predictor X, so

$$\text{var}(\varepsilon_i) = k^2 x_i^2, \quad k > 0.$$

In the *weighted least squares* (WLS) method, we can stabilize the variance by dividing both sides of Equation A.1 by x_i:

$$\frac{y_i}{x_i} = \frac{\beta_0}{x_i} + \beta_1 + \frac{\varepsilon_i}{x_i}, \tag{A.2}$$

then $\text{var}\left(\frac{\varepsilon_i}{x_i}\right) = k^2$, i.e., it is now *stabilized*.

Example: WLS applied 'manually'

Consider a simulated example of a linear model $y = 3 - 2x$ with noise, which is a function of x.

```
set.seed(111)
k = 0.5
n = 100
x <- rnorm(n, 0, 5)
y <- 3 - 2 * x + rnorm(n, 0, k*x^2)
```

The coefficients estimated using ordinary least squares (OLS):

```
fit_ols <- lm(y ~ x)
summary(fit_ols)
```

```
#>
#> Call:
#> lm(formula = y ~ x)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -47.41  -8.01  -2.32   1.58 286.47
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)    6.131      3.402    1.80    0.075 .
```

```
#> x             -3.571      0.639   -5.59    2e-07 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 34 on 98 degrees of freedom
#> Multiple R-squared:  0.242,  Adjusted R-squared:  0.234
#> F-statistic: 31.3 on 1 and 98 DF,  p-value: 2.04e-07
```

Based on Figure A.1, the OLS assumption of homoskedasticity is violated, because the observations deviate farther from the regression line at its ends (i.e., the variability of regression residuals is higher at the low and high values of the predictor).

```
p1 <- ggplot(data.frame(x, y), aes(x = x, y = y)) +
    geom_abline(intercept = 3, slope = -2, col = "gray50", lwd = 1.5) +
    geom_abline(intercept = fit_ols$coefficients[1],
                slope = fit_ols$coefficients[2], lty = 2) +
    geom_point()
p2 <- ggplot(data.frame(x, y), aes(x = x, y = rstandard(fit_ols))) +
    geom_hline(yintercept = 0, col = "gray50") +
    geom_point() +
    xlab("x") +
    ylab("Standardized residuals")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

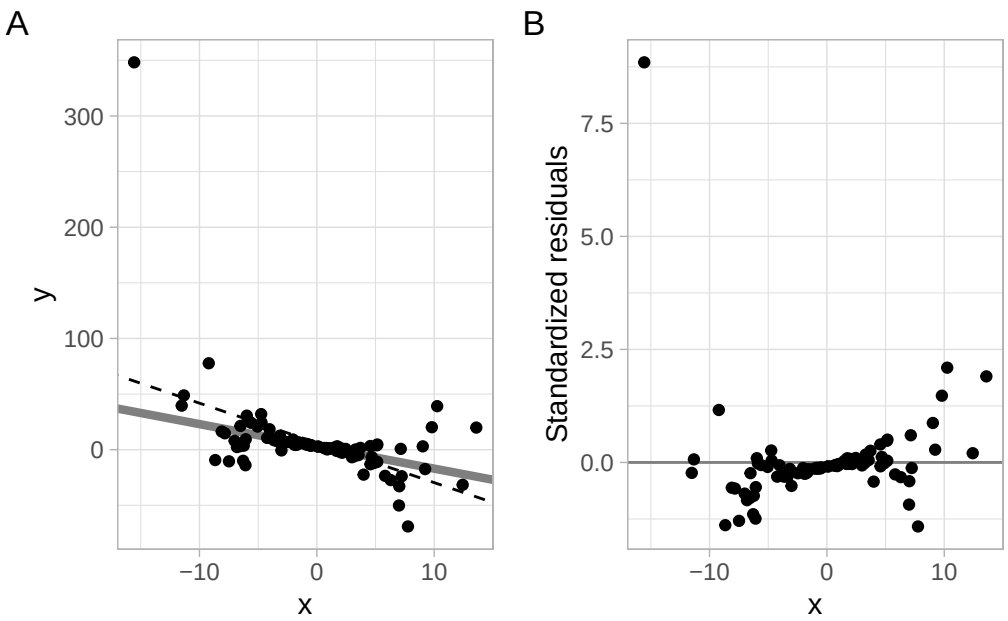


Figure A.1.: Simulated data example with heteroskedasticity. The gray line represents the underlying model; the dashed line is obtained from the OLS fit.

To stabilize the variance 'manually,' transform the variables according to Equation A.2 and refit the model:

```
Y.t <- y/x
X.t <- 1/x
fit_wls <- lm(Y.t ~ X.t)
summary(fit_wls)
```

```
#>
#> Call:
#> lm(formula = Y.t ~ X.t)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -20.021  -0.635   0.251   1.322   5.676
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   -2.154      0.293   -7.36  5.7e-11 ***
#> X.t            3.008      0.168   17.95  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.9 on 98 degrees of freedom
#> Multiple R-squared:  0.767,  Adjusted R-squared:  0.764
#> F-statistic:  322 on 1 and 98 DF,  p-value: <2e-16
```

Check Equation A.2 to see the correspondence of the coefficients, see the results in Figure A.2.

```
p1 <- ggplot(data.frame(x, y), aes(x = x, y = y)) +
    geom_abline(intercept = 3, slope = -2, col = "gray50", lwd = 1.5) +
    geom_abline(intercept = fit_wls$coefficients[2],
                slope = fit_wls$coefficients[1], lty = 2) +
    geom_point()
p2 <- ggplot(data.frame(x, y), aes(x = x, y = rstandard(fit_wls))) +
    geom_hline(yintercept = 0, col = "gray50") +
    geom_point() +
    xlab("x") +
    ylab("Standardized residuals")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

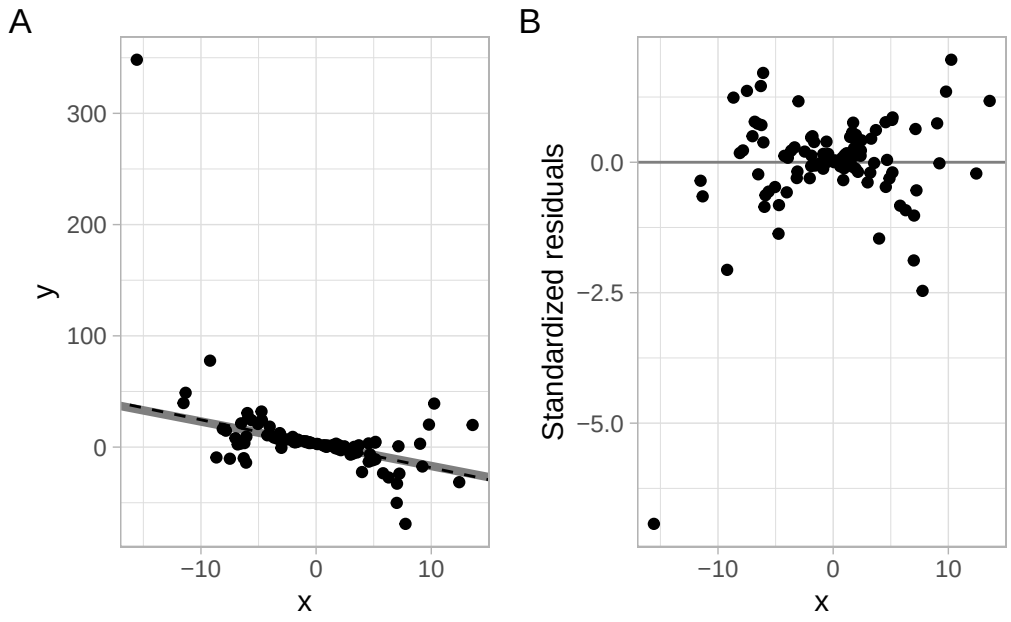


Figure A.2.: Simulated data example with heteroskedasticity. The gray line represents the underlying model; the dashed line is obtained from the WLS fit.

Instead of minimizing the residual sum of squares (using the original or transformed data in Equation A.1 and Equation A.2),

$$RSS(\beta) = \sum_{i=1}^{n} (y_i - x_i\beta)^2,$$

we minimize the *weighted sum of squares*, where w_i are the weights:

$$WSS(\beta; w) = \sum_{i=1}^{n} w_i (y_i - x_i\beta)^2.$$

This includes OLS as the special case when all the weights $w_i = 1$ $(i = 1, \dots, n)$. In the example above, $w_i = 1/x_i^2$.

In matrix form,

$$\hat{\beta} = (X^\top W X)^{-1} X^\top W Y. \tag{A.3}$$

To apply Equation A.3 in R, specify the argument `weights`, and remember to take an inverse. Note that the coefficients are now labeled as expected.

```
fit_wls2 <- lm(y ~ x, weights = 1/x^2)
summary(fit_wls2)
```

```
#>
#> Call:
#> lm(formula = y ~ x, weights = 1/x^2)
```

```
#>
#> Weighted Residuals:
#>    Min     1Q Median     3Q    Max
#> -7.122 -0.842  0.018  1.238 20.021
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)    3.008      0.168   17.95  < 2e-16 ***
#> x             -2.154      0.293   -7.36  5.7e-11 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.9 on 98 degrees of freedom
#> Multiple R-squared:  0.356,  Adjusted R-squared:  0.349
#> F-statistic: 54.2 on 1 and 98 DF,  p-value: 5.72e-11
```

Chatterjee and Hadi (2006) in Chapter 7 consider two more cases for applying WLS, both related to grouping. We skip those cases for now and revisit our data example from Chapter 1.

Example: Dishwasher shipments WLS model

First, use OLS to estimate the simple linear regression exploring dishwasher shipments (DISH) and private residential investments (RES) for several years.

```
D <- read.delim("data/dish.txt") %>%
    rename(Year = YEAR)
modDish_ols <- lm(DISH ~ RES, data = D)
```

The plot in Figure A.3 indicates that the variance might be decreasing with higher investments.

```
ggplot(D, aes(x = RES, y = rstandard(modDish_ols))) +
    geom_hline(yintercept = 0, col = "gray50") +
    geom_point() +
    xlab("Residential investments") +
    ylab("Standardized residuals")
```

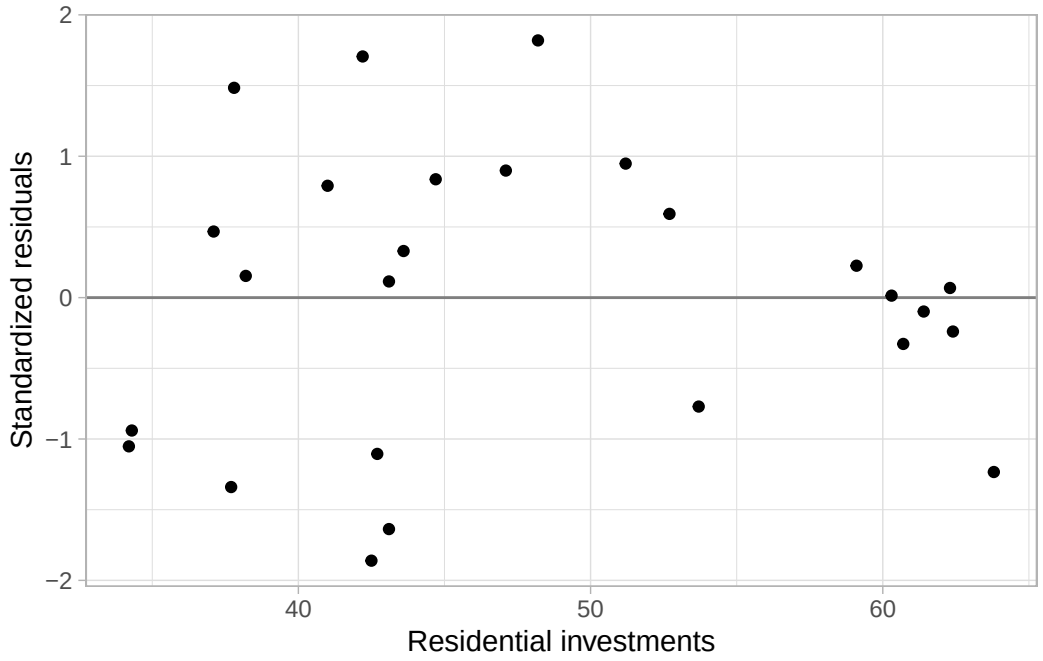


Figure A.3.: OLS residuals vs. the predictor.

Apply the WLS:

```
modDish_wls <- lm(DISH ~ RES, data = D, weights = RES^2)
```

In Figure A.4 we see minor changes in the slope (better fit?).

```
p1 <- ggplot(D, aes(x = RES, y = DISH)) +
    geom_abline(intercept = modDish_wls$coefficients[1],
                slope = modDish_wls$coefficients[2], lty = 2) +
    geom_abline(intercept = modDish_ols$coefficients[1],
                slope = modDish_ols$coefficients[2],
                col = "gray50") +
    geom_point() +
    xlab("Residential investments") +
    ylab("Dishwasher shipments")
p2 <- ggplot(D, aes(x = RES, y = rstandard(modDish_wls))) +
    geom_hline(yintercept = 0, col = "gray50") +
    geom_point() +
    xlab("Residential investments") +
    ylab("Standardized residuals")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

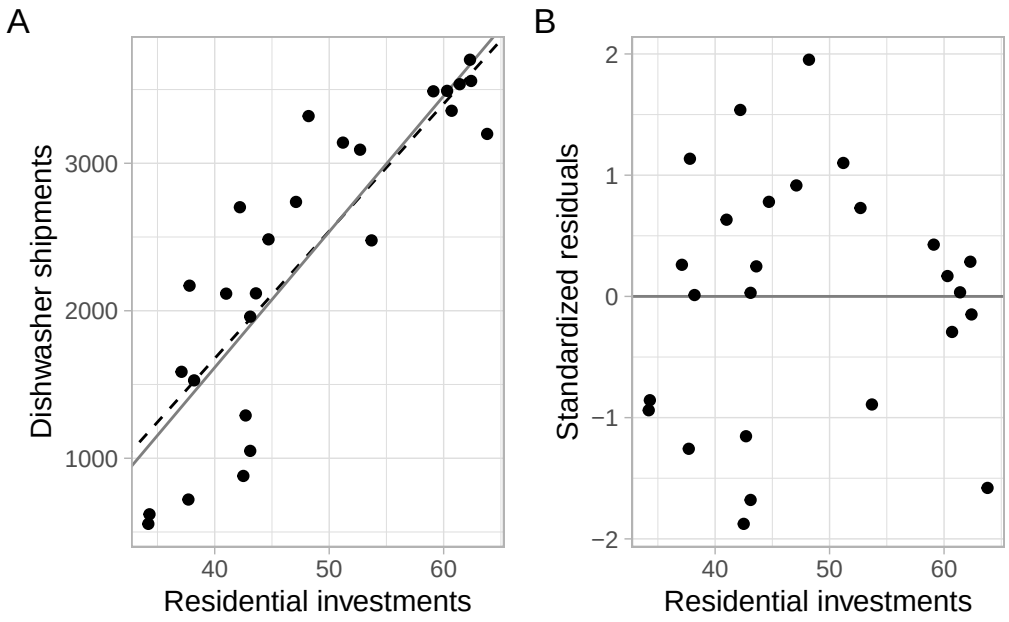


Figure A.4.: The regression fits (OLS – solid line; WLS – dashed line) and the WLS residuals vs. the predictor.

However, the residuals are still autocorrelated, which violates another assumption of the OLS and WLS methods:

```
lawstat::runs.test(rstandard(modDish_wls))
```

```
#>
#>  Runs Test - Two sided
#>
#> data:  rstandard(modDish_wls)
#> Standardized Runs Statistic = -2, p-value = 0.05
```

See Appendix B on the method of generalized least squares (GLS) that allows accounting for autocorrelation in regression modeling.

B. Generalized least squares

Here we use time series data (ordered by t), thus, Equation A.1 will be written with the time indices t as

$$y_t = \beta_0 + \beta_1 x_t + \varepsilon_t, \tag{B.1}$$

where the regression errors at times t and $t-1$ are

$$\begin{aligned} \varepsilon_t &= y_t - \beta_0 - \beta_1 x_t, \\ \varepsilon_{t-1} &= y_{t-1} - \beta_0 - \beta_1 x_{t-1}. \end{aligned} \tag{B.2}$$

An AR(1) model for the errors will yield

$$\begin{aligned} y_t - \beta_0 - \beta_1 x_t &= \rho \varepsilon_{t-1} + w_t, \\ y_t - \beta_0 - \beta_1 x_t &= \rho(y_{t-1} - \beta_0 - \beta_1 x_{t-1}) + w_t, \end{aligned} \tag{B.3}$$

where w_t are uncorrelated errors.

Rewrite it as

$$\begin{aligned} y_t - \rho y_{t-1} &= \beta_0(1-\rho) + \beta_1(x_t - \rho x_{t-1}) + w_t, \\ y_t^* &= \beta_0^* + \beta_1 x_t^* + w_t, \end{aligned} \tag{B.4}$$

where $y_t^* = y_t - \rho y_{t-1}$; $\beta_0^* = \beta_0(1-\rho)$; $x_t^* = x_t - \rho x_{t-1}$. Notice the errors w_t in the final Equation B.4 for the transformed variables y_t^* and x_t^* are uncorrelated.

To get from Equation B.1 to Equation B.4, we can use an iterative procedure by Cochrane and Orcutt (1949) as in the example below.

Example: Dishwasher shipments model accounting for autocorrelation

1. Estimate the model in Equation B.1 using OLS.

```
D <- read.delim("data/dish.txt") %>%
    rename(Year = YEAR)
modDish_ols <- lm(DISH ~ RES, data = D)
```

2. Calculate residuals $\hat{\varepsilon}_t$ and estimate ρ as

$$\hat{\rho} = \frac{\sum_{t=2}^{n} \hat{\varepsilon}_t \hat{\varepsilon}_{t-1}}{\sum_{t=1}^{n} \hat{\varepsilon}_t^2}.$$

```
e <- modDish_ols$residuals
rho <- sum(e[-1] * e[-length(e)]) / sum(e^2)
rho
```

```
#> [1] 0.694
```

3. Calculate transformed variables x_t^* and y_t^* and fit model in Equation B.4.

```
y.star <- D$DISH[-1] - rho * D$DISH[-length(D$DISH)]
x.star <- D$RES[-1] - rho * D$RES[-length(D$RES)]
modDish_ar1 <- lm(y.star ~ x.star)
summary(modDish_ar1)
```

```
#>
#> Call:
#> lm(formula = y.star ~ x.star)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -479.7 -117.8   32.9  120.7  536.1
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)    26.76     130.69    0.20     0.84
#> x.star         50.99       7.74    6.59    1e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 252 on 23 degrees of freedom
#> Multiple R-squared:  0.654,  Adjusted R-squared:  0.639
#> F-statistic: 43.4 on 1 and 23 DF,  p-value: 1.01e-06
```

4. Examine the residuals of the newly fitted equation (Figure B.1) and repeat the procedure, if needed.

```
p1 <- ggplot(D, aes(x = Year, y = modDish_ols$residuals)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    ggtitle("OLS model modDish_ols") +
    ylab("Residuals")
p2 <- ggplot(D[-1,], aes(x = Year, y = modDish_ar1$residuals)) +
    geom_line() +
    geom_hline(yintercept = 0, lty = 2, col = 4) +
    ggtitle("Transformed model modDish_ar1") +
    ylab("Residuals")
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

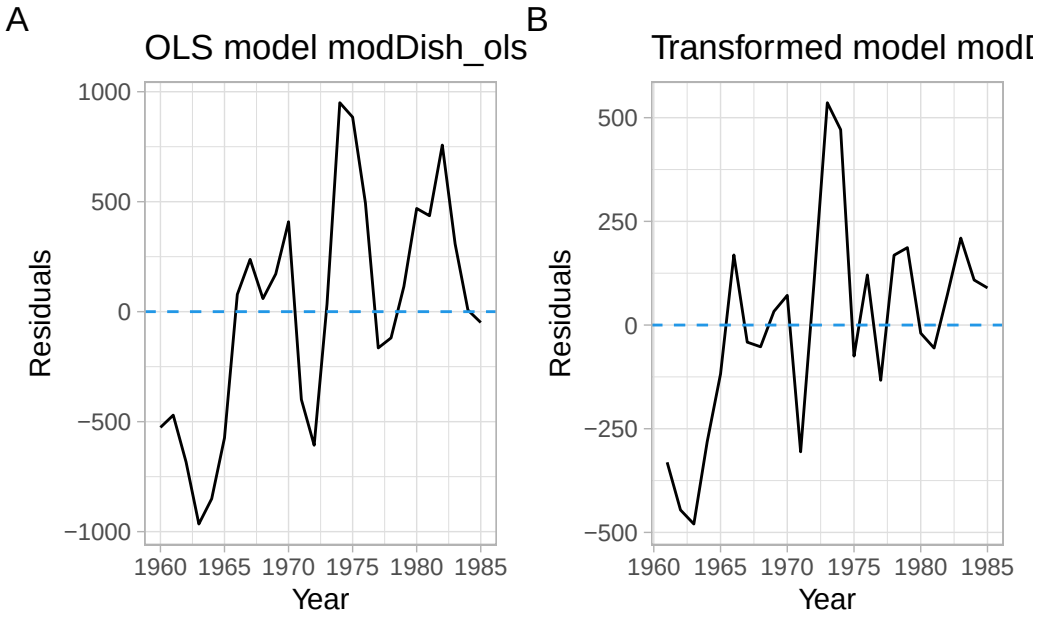


Figure B.1.: Residual plots of the original OLS model and the model transformed to account for autocorrelation in residuals.

Based on the runs test, there is not enough evidence of autocorrelation in the new residuals:

```
lawstat::runs.test(rstandard(modDish_ar1))
```

```
#>
#>  Runs Test - Two sided
#>
#> data:  rstandard(modDish_ar1)
#> Standardized Runs Statistic = -0.6, p-value = 0.5
```

What we have just applied is the method of *generalized least squares* (GLS):

$$\hat{\beta} = \left(X^\top \Sigma^{-1} X\right)^{-1} X^\top \Sigma^{-1} Y, \tag{B.5}$$

where Σ is the covariance matrix. The method of weighted least squares (WLS; Appendix A) is just a special case of the GLS. In the WLS approach, all the off-diagonal entries of Σ are 0.

We can use the function `nlme::gls()` and specify the `correlation` structure to avoid iterating the steps from the previous example manually:

```
modDish_ar1_v2 <- nlme::gls(DISH ~ RES
                            ,correlation = nlme::corAR1(form = ~Year)
                            ,data = D)
summary(modDish_ar1_v2)
```

```
#> Generalized least squares fit by REML
#>   Model: DISH ~ RES
#>   Data: D
#>   AIC BIC logLik
#>   342 347   -167
#>
#> Correlation Structure: AR(1)
#>  Formula: ~Year
#>  Parameter estimate(s):
#> Phi
#>   1
#>
#> Coefficients:
#>              Value Std.Error t-value p-value
#> (Intercept) -137.5   3714140    0.00       1
#> RES           45.7         6    7.35       0
#>
#>  Correlation:
#>     (Intr)
#> RES 0
#>
#> Standardized residuals:
#>       Min        Q1       Med        Q3       Max
#> -0.000249 -0.000014  0.000135  0.000232  0.000338
#>
#> Residual standard error: 3714140
#> Degrees of freedom: 26 total; 24 residual
```

> **Note**
>
> In the function `nlme::gls()` we can also specify `weights` to accommodate heteroskedastic errors, but the syntax differs from the `weights` specification in the function `stats::lm()` (Appendix A). See `?nlme::varFixed`.

C. Synchrony of parametric trends

The problem of detecting joint trend dynamics in time series is essential in a variety of applications, ranging from the analysis of macroeconomic indicators (Vogelsang and Franses 2005; Eun and Lee 2010) to assessing patterns in ice phenology measurements from multiple locations (Latifovic and Pouliot 2007; Duguay et al. 2013) to evaluating yields of financial instruments at various maturity levels (Park et al. 2009) and cell phone download activity at different area codes (Degras et al. 2012).

The extensive research on comparing trend patterns follows two main directions:

1. Testing for joint mean functions.
2. Analysis of joint stochastic trends, which is closely linked to the cointegration notion by Engle and Granger (1987) (Section 11.3).

Here we explore the first direction, that is, assess whether several observed time series follow the same hypothesized parametric trend.

There exist many tests for comparing mean functions, but most of the developed methods assume independent errors. Substantially less is known about testing for joint deterministic trends in a time series framework.

One of the methods developed for time series is by Degras et al. (2012) and Zhang (2013) who extended the integrated square error (ISE) based approach of Vilar-Fernández and González-Manteiga (2004) to a case of multiple time series with weakly dependent (non)stationary errors. For a comprehensive literature review of available methodology for comparing mean functions embedded into independent errors in a time series framework, see Degras et al. (2012) and Park et al. (2014). Most of these methods, however, either focus on aligning only two curves or require us to select multiple hyperparameters, such as the bandwidth, level of smoothness, and window size for a long-run variance function. As mentioned by Park et al. (2014), the choice of such *multiple nuisance parameters* is challenging for a comparison of curves (even under the independent and identically distributed setup) and often leads to inadequate performance, especially in samples of moderate size.

As an alternative, consider an extension of the WAVK test (Section 8.4.2) to a case of multiple time series (Lyubchich and Gel 2016). Let us observe N time series

$$Y_{it} = \mu_i(t/T) + \epsilon_{it},$$

where $i = 1, \dots, N$ (N is the number of time series), $t = 1, \dots, T$ (T is the length of the time series), $\mu_i(u)$ $(0 < u \leqslant 1)$ are unknown smooth trend functions, and the noise ϵ_{it} can be represented as a finite-order AR(p) process or infinite-order AR(∞) process with i.i.d. innovations e_{it}.

We are interested in testing whether these N observed time series have the same trend of some pre-specified smooth parametric form $f(\theta, u)$:

H_0: $\mu_i(u) = c_i + f(\theta, u)$
H_1: there exists i, such that $\mu_i(u) \neq c_i + f(\theta, u)$,

where the reference curve $f(\theta, u)$ with a vector of parameters θ belongs to a known family of smooth parametric functions, and $1 \leqslant i \leqslant N$. For identifiability, assume that $\sum_{i=1}^{N} c_i = 0$. Notice that the hypotheses include (but are not limited to) the special cases of
$f(\theta, u) \equiv 0$ (testing for no trend);
$f(\theta, u) = \theta_0 + \theta_1 u$ (testing for a common linear trend);
$f(\theta, u) = \theta_0 + \theta_1 u + \theta_2 u^2$ (testing for a common quadratic trend).

This hypothesis testing approach allows us to answer the following questions:

- Do trends in temperature (or wind speeds, or precipitation) reproduced by a climate model correspond to the historical observations? I.e., is the model generally correct?
- Do different instruments (sensors) capture changes similarly, or deviate, for example, due to aging of some of the instruments?
- Do trends estimated at different locations (Canada and USA, lower and mid-troposphere, etc.) follow some hypothesized global trend?

Test the null hypothesis by following these steps (Lyubchich and Gel 2016):

1. Estimate the joint hypothetical trend $f(\theta, u)$ using the aggregated sample $\left\{\overline{Y}_{\cdot t}\right\}_{t=1}^{T}$ (i.e., a time series obtained by averaging across all N time series).
2. For each time series, subtract the estimated trend, then apply the autoregressive filter to obtain residuals $\hat{e}_{it}$, which under the H_0 behave asymptotically like the independent and identically distributed e_{it}:

$$\begin{aligned}\hat{e}_{it} &= \hat{\epsilon}_{it} - \sum_{j=1}^{p_i(T)} \hat{\phi}_{ij}\hat{\epsilon}_{i,t-j} \\ &= \left\{Y_{it} - f(\hat{\theta}, u_t)\right\} - \left\{\sum_{j=1}^{p_i(T)} \hat{\phi}_{ij} Y_{i,t-j} - \sum_{j=1}^{p_i(T)} \hat{\phi}_{ij} f(\hat{\theta}, u_{t-j})\right\}.\end{aligned}$$

3. Construct a sequence of N statistics $\mathrm{WAVK}_1(k_1), \ldots, \mathrm{WAVK}_N(k_N)$. Then, the synchrony test statistic is
$$S_T = \sum_{i=1}^{N} k_i^{-1/2} \mathrm{WAVK}_i(k_i),$$
where k_i is the local window size for the WAVK statistic.
4. Estimate the variance of $\hat{e}_{it}$, e.g., using the robust difference-based estimator by Rice (1984):
$$s_i^2 = \frac{\sum_{t=2}^{T} (\hat{e}_{it} - \hat{e}_{i,t-1})^2}{2(T-1)}.$$
5. Simulate BN times T-dimensional vectors e_{iT}^* from the multivariate normal distribution $MVN\left(0, s_i^2 I\right)$, where B is the number of bootstrap replications, I is a $T \times T$ identity matrix.
6. Compute B bootstrapped statistics on e_{iT}^*:
$$S_T^* = \sum_{i=1}^{N} k_i^{-1/2} \mathrm{WAVK}_i^*(k_i).$$

7. The bootstrap p-value for testing the H_0 is the proportion of $|S_T^*|$ that exceed $|S_T|$.

See the application of both the WAVK and synchrony tests in Lyubchich (2016).

If the null hypothesis is rejected, the method does not tell, however, what was the reason, and which particular time series caused the rejection of the H_0. One can remove the time series (or several time series at once) with the largest WAVK statistic(s) and apply the test again, although repeated testing increases the probability of Type I error. For an application of this method in trend clustering, see this vignette.

Example: CMIP5 vs. observations

Replicate the test for synchrony of trends found in two time series (Lyubchich 2016):

- a multi-model average of temperatures from the 5th phase of the Coupled Model Intercomparison Project (CMIP5) and
- observed global temperature anomalies relative to the base period of 1981–2010.

```
D <- read.csv("data/CMIP5.csv") %>%
    filter(1948 <= Year & Year <= 2013) %>%
    mutate(Temp_CMIP5 = Temp_CMIP5 - 273.15)
```

See Figure C.1 showing the time series plots after converting the CMIP data to degrees Celsius.

```
p1 <- D %>% ggplot(aes(x = Year, y = Temp_CMIP5)) +
    geom_line()
p2 <- D %>% ggplot(aes(x = Year, y = Temp_obs)) +
    geom_line()
p1 + p2 +
    plot_annotation(tag_levels = 'A')
```

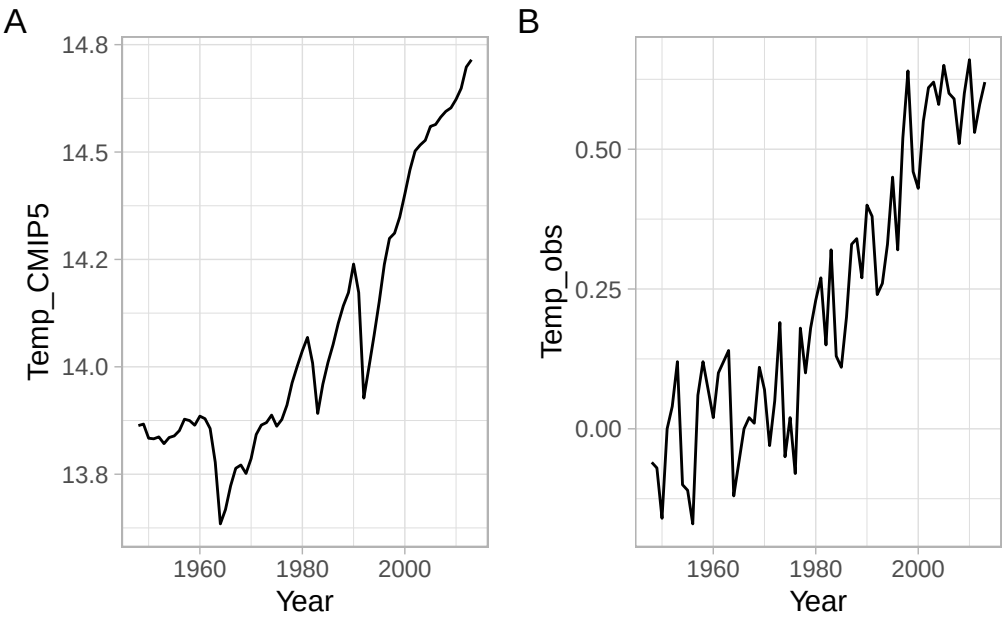


Figure C.1.: Global annual mean temperature (°C) in 1948–2013: CMIP5 multi-model average and observed anomalies relative to the base period of 1981–2010.

Test the synchrony of parametric *linear* trends in these time series:

```
set.seed(123)
funtimes::sync_test(D[, c("Temp_CMIP5", "Temp_obs")] ~ t)
```

```
#>
#>  Nonparametric test for synchronism of parametric trends
#>
#> data:  D[, c("Temp_CMIP5", "Temp_obs")]
#> Test statistic = 0.04, p-value = 0.03
#> alternative hypothesis: common trend is not of the form D[, c("Temp_CMIP5", "Temp_obs")] ~
#> sample estimates:
#> $common_trend_estimates
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)    -1.57     0.0962   -16.4 1.44e-24
#> t               3.10     0.1647    18.8 8.39e-28
#>
#> $ar.order_used
#>          Temp_CMIP5 Temp_obs
#> ar.order          1        1
#>
#> $Window_used
#>        Temp_CMIP5 Temp_obs
#> Window          3        3
```

```
#>
#> $all_considered_windows
#>  Window Statistic p-value Asympt. p-value
#>       3    0.0406   0.034          0.0246
#>       4    0.0373   0.042          0.0386
#>       6    0.0413   0.020          0.0222
#>
#> $wavk_obs
#> [1] 0.0143 0.0263
```

The p-value below the usual significance level $\alpha = 0.05$ allows us to reject the null hypothesis, however, as Lyubchich (2016) pointed out, the decision would differ if more confidence is required (e.g., $\alpha = 0.01$). Note that the p-value of 0.012 reported by Lyubchich (2016) differs from the one reported above due to the function settings and randomness due to the bootstrapping. We should save the random number generator state with `set.seed()` for replicability (so the test results are exactly the same every time the test is applied), and use a larger number of bootstrap replications `B` for consistency (so the test leads to the same conclusions when the function `set.seed()` is not used).
Now test the synchrony of parametric *quadratic* trends in these time series:

```
set.seed(123)
funtimes::sync_test(D[, c("Temp_CMIP5", "Temp_obs")] ~ poly(t, 2))
```

```
#>
#>  Nonparametric test for synchronism of parametric trends
#>
#> data:  D[, c("Temp_CMIP5", "Temp_obs")]
#> Test statistic = 0.05, p-value = 0.002
#> alternative hypothesis: common trend is not of the form D[, c("Temp_CMIP5", "Temp_obs")] ~
#> sample estimates:
#> $common_trend_estimates
#>              Estimate Std. Error   t value Pr(>|t|)
#> (Intercept) -1.77e-15     0.0324 -5.45e-14 1.00e+00
#> poly(t, 2)1  7.27e+00     0.2632  2.76e+01 6.20e-37
#> poly(t, 2)2  2.28e+00     0.2632  8.65e+00 2.62e-12
#>
#> $ar.order_used
#>          Temp_CMIP5 Temp_obs
#> ar.order          2        0
#>
#> $Window_used
#>        Temp_CMIP5 Temp_obs
#> Window          3        3
#>
#> $all_considered_windows
#>  Window Statistic p-value Asympt. p-value
#>       3    0.0518   0.002        0.000282
```

```
#>       4    0.0350   0.028        0.014236
#>       6    0.0222   0.070        0.120094
#>
#> $wavk_obs
#> [1] -0.000673  0.052504
```

Note these results differ substantially based on the window used for computing the WAVK statistic. The function `funtimes::sync_test()` automatically selects the optimal window based on the heuristic approach of comparing distances between bootstrap distributions (Lyubchich 2016; Lyubchich and Gel 2016).

D. Time Series Clustering

D.1. Overview

Time series clustering groups multiple time series based on similarity of their trend dynamics or value profiles. This appendix covers the BICC/CWindowCluster method (**Ciampi:etal:2010?**) and iterative synchronism testing via the `funtimes` R package.

D.2. Methods

Slide-level clustering — For a data domain $[\alpha, \beta]$ and $\delta \in [0, 1]$, the δ-close measure is:

$$\psi_\delta(x_1, x_2) = \begin{cases} 1 & \frac{\|x_1 - x_2\|_1}{\beta - \alpha} \leq \delta \\ 0 & \text{otherwise} \end{cases} \tag{D.1}$$

Time series buckets B_u and B_v are merged into one cluster when:

$$\sum_{i=1}^{p} \psi_\delta(B_u[i], B_v[i]) \geq \theta \times p$$

where $\theta \in [0, 1]$ controls the homogeneity threshold and p is the number of snapshots per slide.

Window-level clustering — Time series are grouped together if they appear in the same slide cluster more than $\varepsilon \times w$ times across w slides.

D.3. Example

> **Note**
>
> Content to be added. See `funtimes::CWindowCluster()` for implementation.

E. Analysis of precipitation extremes and climate projections

Text

F. Practice exercises

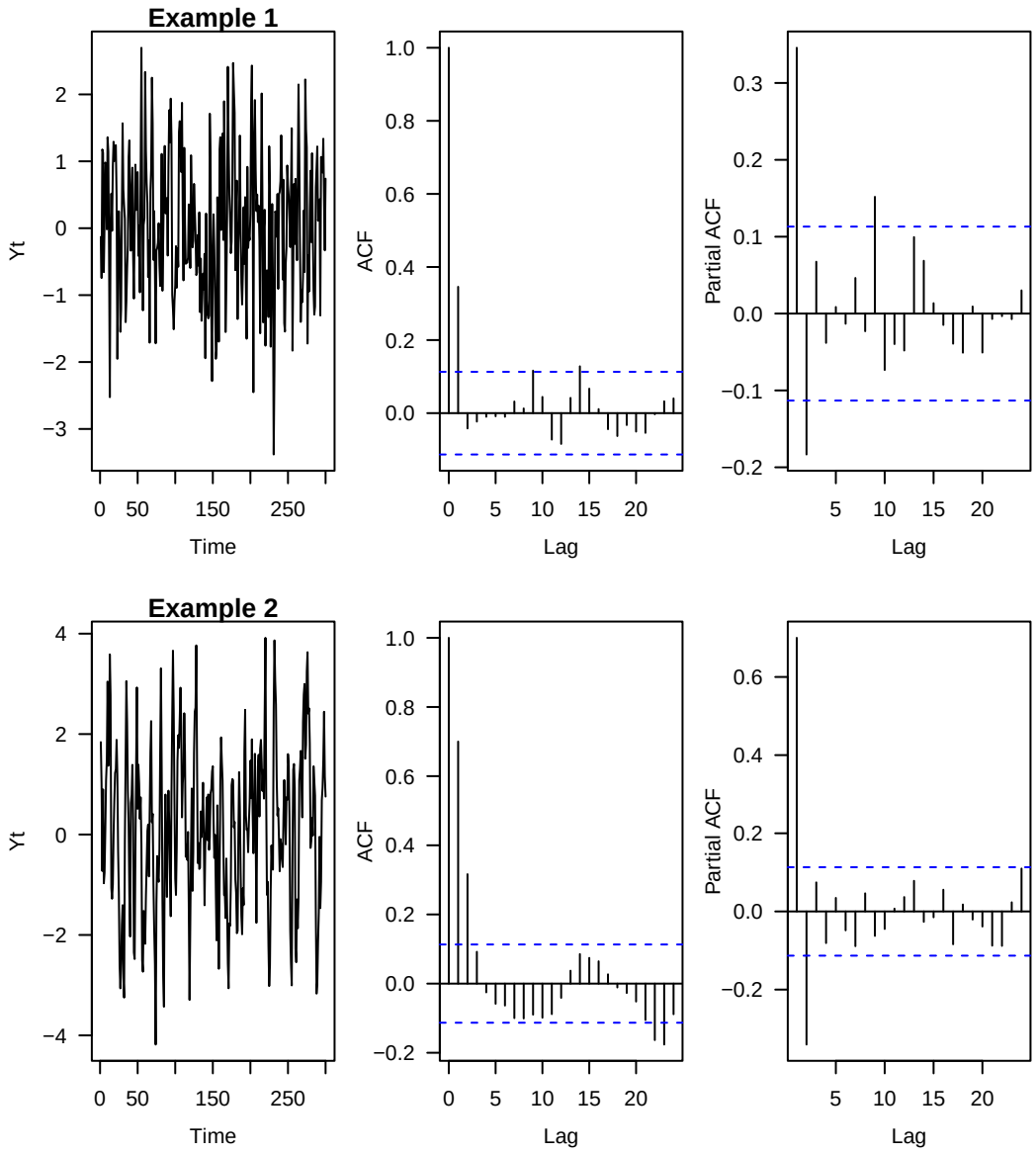
F.1. Intro practice

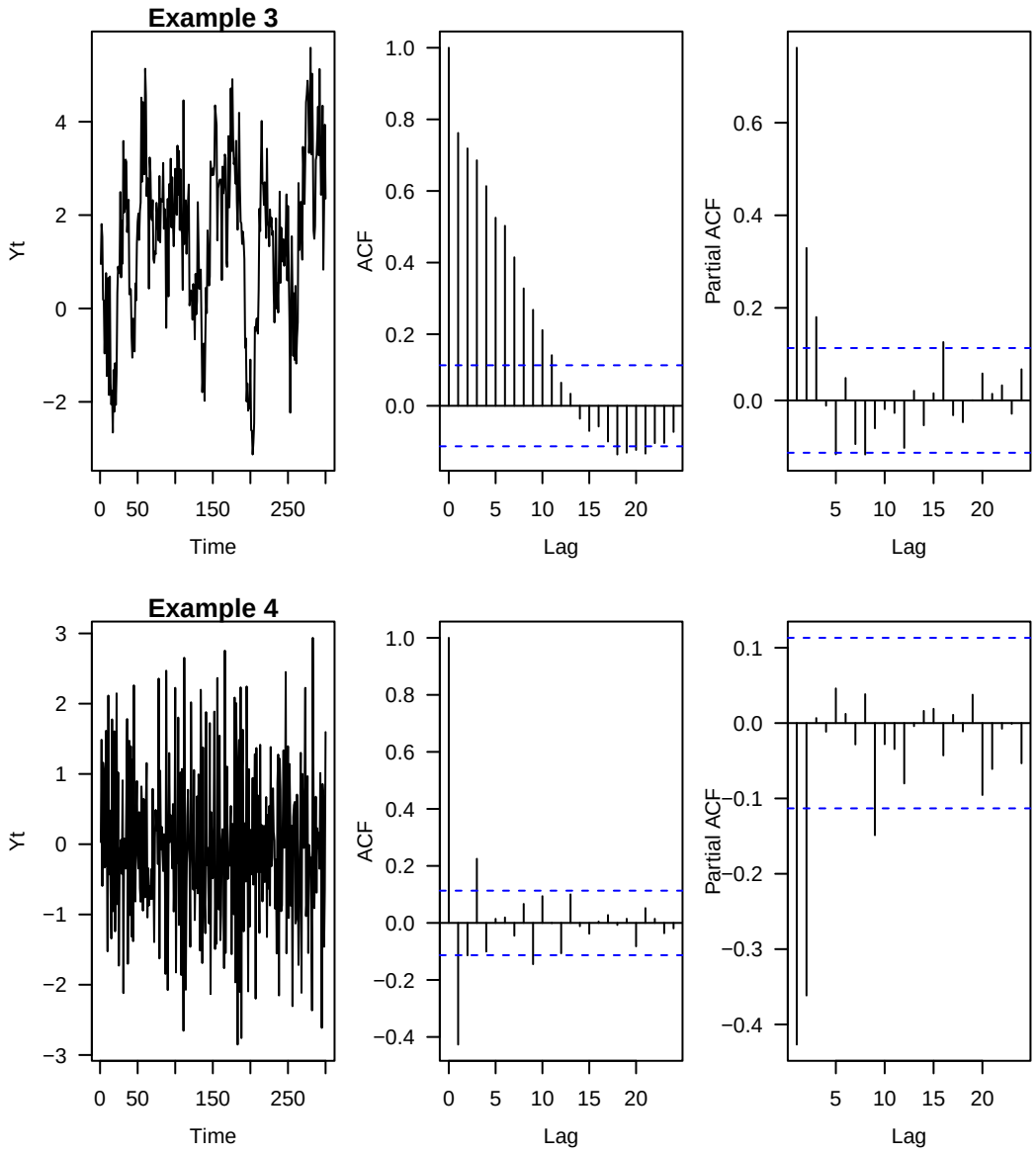
Answer whether each of these statements is true or false.

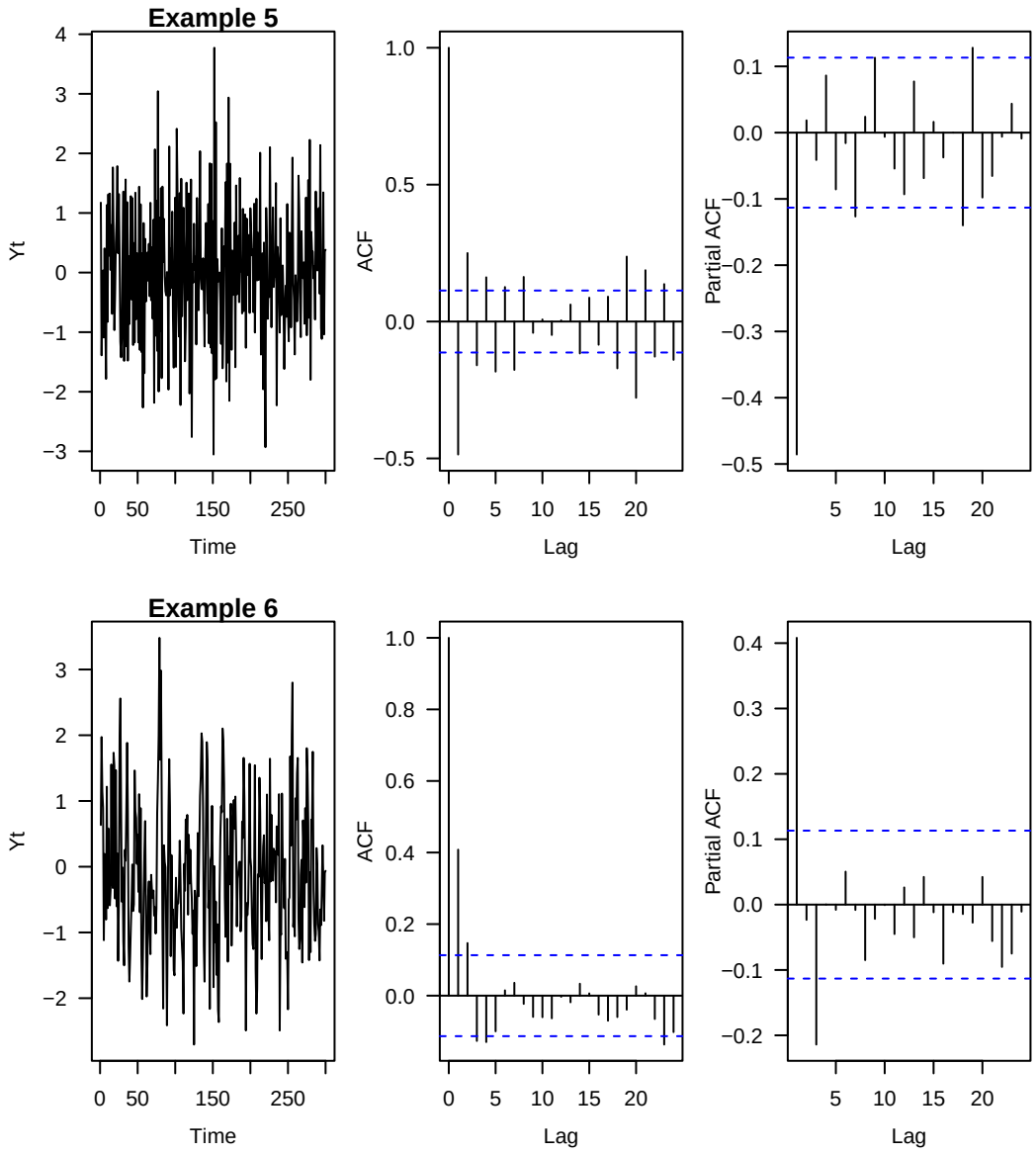
1. The highest autocorrelation is observed when each next observation is exactly the same as the previous.
2. 'Time series is autocorrelated' means there is a trend.
3. Autocorrelation goes away if we smooth the data, for example, with a moving average.
4. 'Random variables X and Y are uncorrelated' means X and Y are independent.
5. Time series is an uninterrupted sequence of observations. Missing observations break the sequence into multiple separate time series.
6. The most appropriate statistical tool to detect a trend is the simple Student's t-test.
7. If there is no autocorrelation at the first lag, i.e., $\text{cor}(X_t, X_{t-1}) = 0$, then $\text{cor}(X_t, X_{t-2}) = 0$.
8. Prediction mean absolute error (PMAE) measures the quality of point forecasts, whereas prediction mean squared error (PMSE) measures the quality of interval forecasts.
9. If a time series X_t $(t = 1, \dots, T)$ is stationary, then all predictions for times $T + 1$, $T + 2$, ... are the same.
10. White noise is a sequence of weakly correlated random variables.

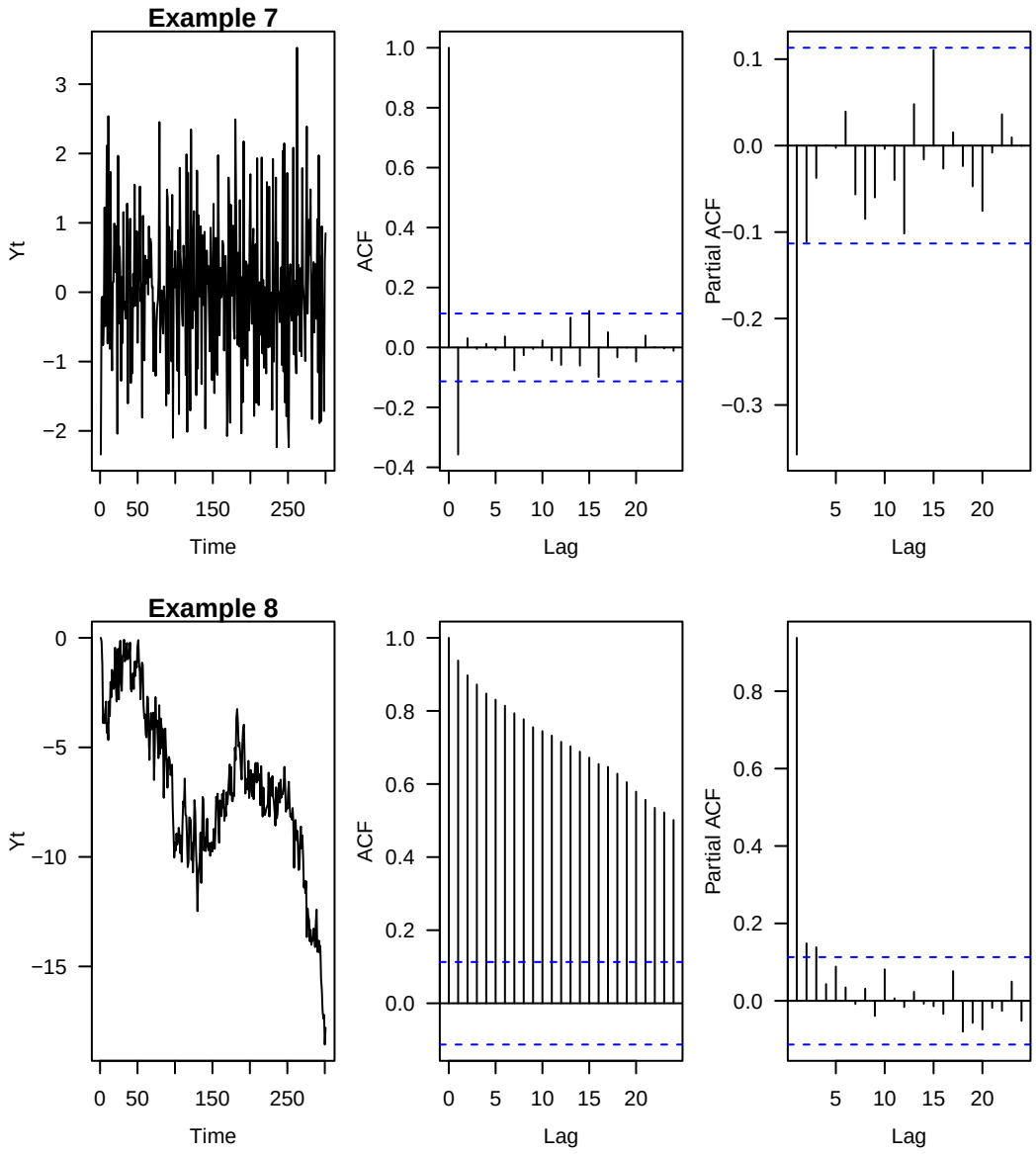
F.2. ARMA practice

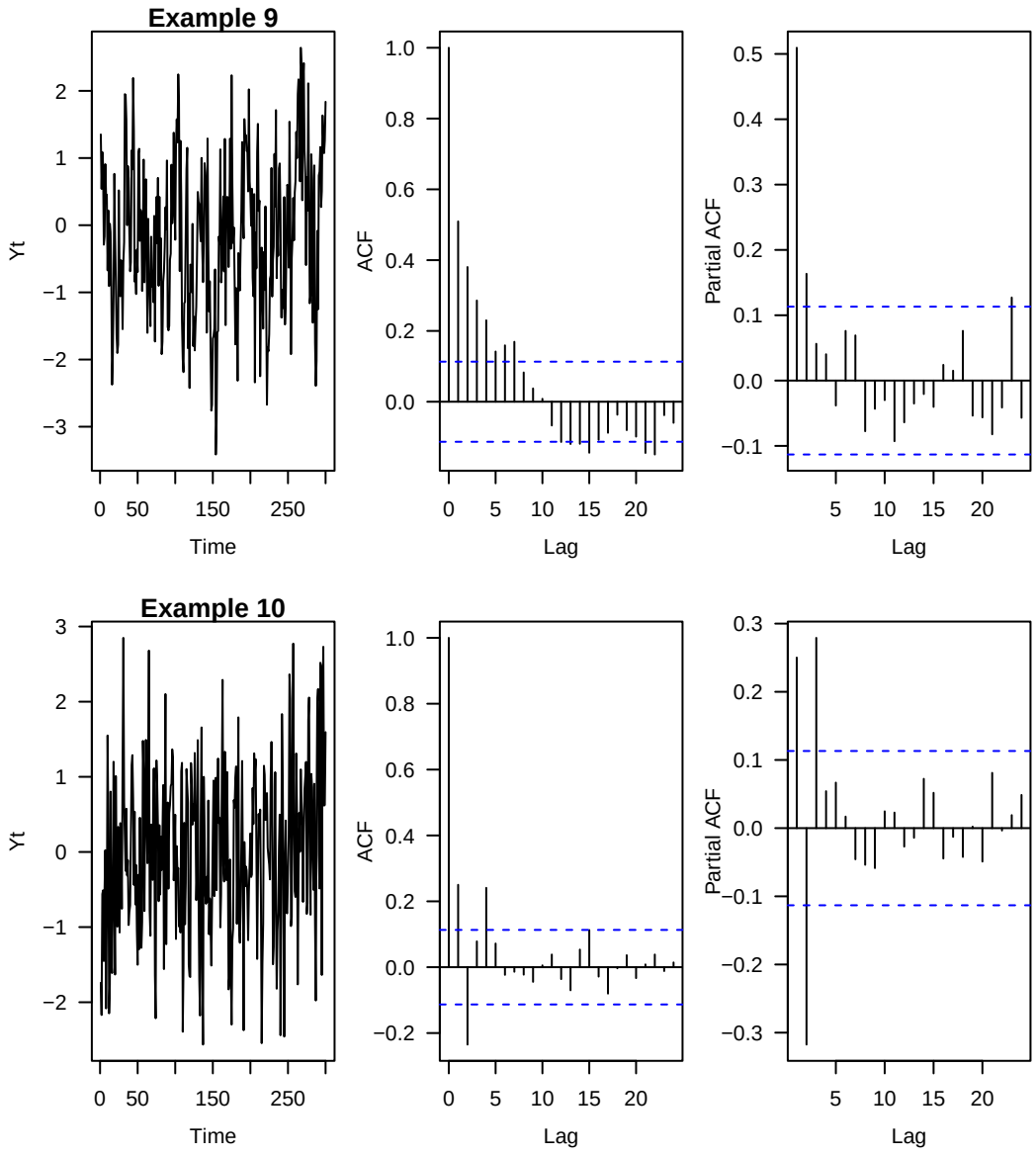
Below are several examples of time series with their ACF and PACF plots. For each example time series, use the plots to decide whether an ARMA(p, q) model is appropriate, and if so, suggest the orders p and q. Use **?@tbl-arma** for help.


Example 1
Yt
Time
ACF
Lag
Partial ACF
Lag
Example 2
Yt
Time
ACF
Lag
Partial ACF
Lag


Example 3
Yt
Time
ACF
Lag
Partial ACF
Lag
Example 4
Yt
Time
ACF
Lag
Partial ACF
Lag


Example 5
Yt
Time
ACF
Lag
Partial ACF
Lag
Example 6
Yt
Time
ACF
Lag
Partial ACF
Lag


Example 7
Yt
Time
ACF
Lag
Partial ACF
Lag
Example 8
Yt
Time
ACF
Lag
Partial ACF
Lag



F.3. Trend practice

Answer whether each of these statements is true or false.

1. If ACF values at the first ten lags are statistically significant, then the time series is not stationary.
2. If X_t is an ARIMA(p, d, q) process, then $(1 - B)^d X_t$ is an ARMA(p, q) process.
3. Slowly decaying ACF is a sign of nonstationarity.
4. If the hypothesis H_1 of a linear trend was accepted for the series U_t, $t = 1, \dots, T$, it will be also accepted for the subsets $U_{t'}$, where $t' = j, \dots, k$; $j < k$, and $j, k < T$.
5. Unit root tests can be applied to determine the appropriate order of differencing d.
6. A time series that exhibits a quadratic-looking trend can be made stationary (detrended) using the Box–Cox transformation with the power parameter $\lambda = 2$.

7. ARIMA(0,1,0) model is a random walk.
8. For the backshift operator B, $(1-B)^d X_t = (1-B^d)X_t$.
9. A linear time trend can be eliminated by differencing the time series once or twice.
10. Trend functions (e.g., $X_t = 0.35 + 0.11t + \epsilon_t$, where ϵ_t are uncorrelated errors) express the changes in the process X_t caused by time.
11. Time series should be differenced just enough times to remove a stochastic trend. Differencing too many times leads to problems.
12. Autocorrelation in observations affects results of the t-test and Mann–Kendall test.
13. The Mann–Kendall test focuses on a more general class of trends than the t-test does.
14. The null hypothesis of the augmented Dickey–Fuller test is no unit root (stationarity).
15. ARIMA(p, d, q) is a difference-stationary process.
16. Bootstrapping allows us to replicate the finite-sample distribution of the test statistic.
17. To detrend a time series, the difference operator should be applied with the same lag(s) at which the sample ACF has statistically significant values.
18. One of the correct ways to run regression on the time series Y_t and X_t with trends is to detrend these time series before fitting the regression model.
19. In practice, it is seldom necessary to go beyond second-order differences for detrending a time series.

Software

Examples were calculated in R version 4.5.3 (2026-03-11) with an effort to use the most recent versions of R packages.

The codes load silently only a few packages:

```
library(dplyr)
library(ggplot2)
library(patchwork)
theme_set(theme_light())
```

All other packages are named before the function as in `forecast::ggAcf()` (this code calls the function `ggAcf()` from the package `forecast`) or called immediately before the necessary function use:

```
library(fable)
m <- as_tsibble(Y) %>%
    model(ARIMA(Y ~ 1, ic = "bic"))
report(m)
```

The R packages used in this book include (in alphabetic order):

- `astsa` (Stoffer 2026)
- `downlit` (Wickham 2025)
- `dplR` (Bunn et al. 2025)
- `dplyr` (Wickham et al. 2026b)
- `dynlm` (Zeileis 2019)
- `Ecdat` (Croissant and Graves 2025)
- `fable` (O'Hara-Wild et al. 2026a)
- `feasts` (O'Hara-Wild et al. 2026b)
- `fGarch` (Wuertz et al. 2025)
- `FinTS` (Graves 2024)
- `fma` (Hyndman 2023)
- `forecast` (Hyndman et al. 2026)
- `funtimes` (Lyubchich et al. 2025)
- `gamlss` (Stasinopoulos and Rigby 2025)
- `gamlss.ggplots` (**R-gamlss.ggplots?**)
- `gamlss.util` (**R-gamlss.util?**)
- `GGally` (Schloerke et al. 2025)
- `ggplot2` (Wickham et al. 2026a)
- `ggpubr` (Kassambara 2026)

- `Kendall` (McLeod 2025)
- `knitr` (Xie 2025)
- `lawstat` (Gastwirth et al. 2023)
- `lmtest` (Hothorn et al. 2022)
- `lomb` (Ruf 2024)
- `mgcv` (Wood 2025)
- `mgcViz` (Fasiolo and Nedellec 2025)
- `nlme` (Pinheiro et al. 2025)
- `oce` (Kelley and Richards 2024)
- `patchwork` (Pedersen 2025)
- `plotly` (Sievert et al. 2026)
- `pracma` (Borchers 2025)
- `randtests` (Caeiro and Mateus 2024)
- `readr` (Wickham et al. 2026c)
- `rmarkdown` (Allaire et al. 2026)
- `signal` (Ligges et al. 2024)
- `tseries` (Trapletti and Hornik 2026)
- `TSstudio` (Krispin 2023)
- `urca` (Pfaff 2024)
- `xml2` (Wickham et al. 2026d)